Guide to Creating Modular Procedures on VAX/VMS

Order Number: AA–FB84A–TE

July 1985

This document describes how to design and code procedures that will conform to the VAX/VMS Modular Programming Standard. It also describes how to insert modules in an object module library, shareable image, or shareable image library.

Revision/Update Information: This is a new manual.

Software Version: VAX/VMS Version 4.2

digital equipment corporation
maynard, massachusetts

July 1985

The postpaid READER'S COMMENTS form on the last page of this document requests the user's critical evaluation to assist in preparing future documentation.

ZK–2774

HOW TO ORDER ADDITIONAL DOCUMENTATION
DIRECT MAIL ORDERS

USA & PUERTO RICO*

Digital Equipment Corporation
P.O. Box CS2008
Nashua, New Hampshire
03061

CANADA

Digital Equipment of Canada Ltd.
100 Herzberg Road
Kanata, Ontario K2K 2A6
Attn: Direct Order Desk

INTERNATIONAL

Digital Equipment Corporation
PSG Business Manager
c/o Digital's local subsidiary or approved distributor

In Continental USA and Puerto Rico call 800-258-1710.

In New Hampshire, Alaska, and Hawaii call 603-884-6660.

In Canada call 800-267-6215 (in Ottawa-Hull 613-234-7726).

*Any prepaid order from Puerto Rico must be placed with the local Digital subsidiary (809-754-7575).

Internal orders should be placed through the Software Distribution Center (SDC), Digital Equipment Corporation, Northboro, Massachusetts 01532.

This document was prepared using an in-house documentation production system. All page composition and make-up was performed by TEX, the typesetting system developed by Donald E. Knuth at Stanford University. TEX is a registered trademark of the American Mathematical Society.

Contents

EXAMPLES

FIGURES

TABLES

Preface

This manual is a tutorial guide to designing and coding modular procedures written in any VAX language. You can use these procedures for general programming. You can also include them in a procedure library, such as an object module library, shareable image, or shareable image library.

This guide includes required and optional modular programming techniques, recommended style, and a description of how to install modular procedures in procedure libraries.

A procedure is modular if it follows rules and principles that permit it to be successfully linked with other procedures that follow the same rules and principles. The *Guide to Creating Modular Procedures on VAX/VMS* tells you how to follow the VAX/VMS Modular Programming Standard (which summarizes these rules and principles). Following this standard will improve program reliability and reduce maintenance effort.

Intended Audience

This manual is intended for advanced system and applications programmers who are already familiar with VAX/VMS system concepts. Readers should be familiar with the VAX/VMS operating system and proficient in at least one supported language.

Structure of This Document

All chapters in this manual are tutorial. The general structure of this manual is based on the DIGITAL version of the software life cycle. All information is presented within this construct.

- Chapter 1 provides an overview of the software life cycle model used in this book. It also defines what is meant by the term "procedure", and provides an overview of procedures that already exist on the VAX/VMS operating system. The layered products that you might find useful in procedure development are also discussed in Chapter 1.

- Chapter 2 provides information on design. Topics covered include: organizing new applications, designing a modular procedure interface, using system resources, using input /output, writing internal documentation, and planning for signaling and condition handling.
- Chapter 3 presents general coding guidelines and information on initializing modular procedures. It also discusses guidelines for invoking optional user-supplied action routines, and writing AST-reentrant code.
- Chapter 4 describes methods used to test procedures for modularity, language-independence, and reentrancy. This chapter also provides general information on performance testing and monitoring procedures.
- Chapter 5 teaches you how to create object module libraries, shareable images, and shareable image libraries from your completed procedures. Special attention is given to the transfer vector and the linker options file.
- Chapter 6 provides information about maintenance, such as upward compatibility, regression testing, updating procedures and procedure libraries, and changing the transfer vector or linker options file.
- Appendix A contains the VAX/VMS Modular Programming Standard.
- Appendix B presents the argument characteristics supported by the VAX Procedure Calling and Condition Handling Standard.

Associated Documents

The following documents are associated with this manual:

- *VAX/VMS Run-Time Library Routines Reference Manual*
- *VAX/VMS System Services Reference Manual*
- *VAX-11 Linker Reference Manual*
- All user's guides and reference manuals for all VAX languages

Conventions Used in This Document

Convention	Meaning
RET	A symbol with a one- to six-character abbreviation indicates that you press a key on the terminal, for example, RET .
CTRL/x	The phrase CTRL/x indicates that you must press the key labeled CTRL while you simultaneously press another key, for example, CTRL/C, CTRL/Y, CTRL/O.
$ SHOW TIME 05-JUN-1985 11:55:22	Command examples show all output lines or prompting characters that the system prints or displays in black letters. All user-entered commands are shown in red letters.
$ TYPE MYFILE.DAT . . .	Vertical series of periods, or ellipsis, mean either that not all the data that the system would display in response to the particular command is shown or that not all the data a user would enter is shown.
file-spec, . . .	Horizontal ellipsis indicates that additional parameters, values, or information can be entered.
[logical-name]	Square brackets indicate that the enclosed item is optional. (Square brackets are not, however, optional in the syntax of a directory name in a file specification or in the syntax of a substring specification in an assignment statement.)
quotation marks apostrophes	The term quotation marks is used to refer to double quotation marks (″). The term apostrophe (′) is used to refer to a single quotation mark.

New and Changed Features

This manual has been completely reorganized and rewritten since the previous version. Most noticeably, the material is now presented in a tutorial fashion, following the development of a modular procedure step-by-step through the software life cycle model, from design through maintenance.

The scope of this manual has been expanded to provide information about testing and maintenance, and complete examples have been added to guide the user in the development of modular procedures and procedure libraries on VAX/VMS.

1 Introduction to Modular Procedures

1.1 Following the Software Life Cycle

This manual discusses the development of modular procedures, procedure libraries, shareable images, and shareable image libraries, in terms of the software life cycle.

There are several different versions of the software life cycle that are currently in use. The one that is used in this manual depicts the software life cycle as being broken down into seven stages.

Figure 1–1 shows the seven stages of the software life cycle.

1. The first stage is the concept stage. During the concept stage, the new application is proposed and considered.

2. The requirements/specifications stage is the point at which the goals of the new application are chosen and a requirements/specifications document is written outlining what will be accomplished by the new application.

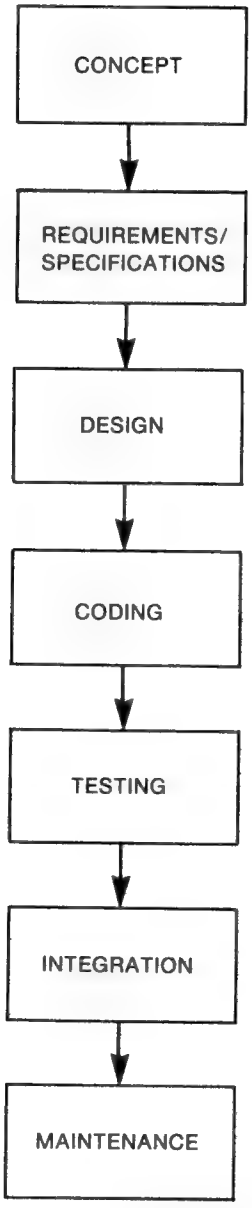
3. At the design stage, detailed plans are made to ensure that the completed code will be modular and accurate, as well as completely meet the goals decided upon at the requirements/specifications stage.

4. During the coding stage, the actual application is coded and internally documented.

5. Once coding is completed, the code is tested to ensure that it is accurate, efficient, and meets the requirements and specifications.

6. At the integration stage, the procedures and applications are grouped into facilities and libraries.

Figure 1–1 The Software Life Cycle



7 The maintenance stage is the point at which the completed applications are updated, enhanced, and modified.

Modularity becomes an important consideration in the design phase of the software life cycle. Because this manual is primarily concerned with modularity, we begin detailed discussion at the design stage, and continue through the coding, testing, integration, and maintenance stages.

1.2 Defining the Modular Procedure

A procedure is a set of related instructions that performs a task. Typically, a procedure is invoked by executing a VAX CALLS or CALLG instruction. If you are using a high level language, the compiler will generate the CALLS or CALLG instruction on your behalf when you use the conventions required by your language to implement a procedure. VAX languages implement procedures as follows:

Ada	A procedure is declared as a PROCEDURE or FUNCTION.
BASIC	A procedure is a main program, subprogram, or FUNCTION.
BLISS	A procedure is declared as a ROUTINE.
COBOL	A procedure is a paragraph (or SECTION) or group of logically successive paragraphs (or SECTIONs) in the Procedure Division.
FORTRAN	A procedure is a main program, subroutine, or function.
MACRO	A procedure begins with an .ENTRY directive and ends with a RET instruction.
PASCAL	A procedure is declared as a PROCEDURE or FUNCTION.
PL/I	A procedure is an external or internal PL/I PROCEDURE or entry point.
RPG II	A procedure is a main program.

A procedure is modular if it follows rules and principles that permit it to be successfully linked together with other procedures that follow the same rules and principles. These rules and principles are summarized in the VAX/VMS Modular Programming Standard, contained in Appendix A of this manual.

This manual describes how to perform a complex task by dividing it into modules and coding each module as a separate procedure. This kind of modular programming offers several advantages over writing a complex program as a single source module.

- You can use any modular procedure in any program.
- You can add a modular procedure to a library at any time.
- You need not rewrite common algorithms every time they are needed for a new program.
- You can divide a complex program into simpler procedures to reduce development time and complexity, and increase reliability.
- You can modify or replace a procedure without modifying the calling program provided that you adhere to the optional guidelines for maintaining upward compatibility in the VAX/VMS Modular Programming Standard.
- You can control process-wide resource allocation.
- You can use different programming languages to write different procedures for a program.

The VAX/VMS Modular Programming Standard also contains guidelines which are recommended rather than required. If you follow these recommendations, you will gain other advantages in addition to modularity. These advantages are as follows:

- Shareable library procedures can save memory space, disk space, and link time.
- AST-reentrant procedures can be called by AST-level procedures.
- Modular procedures that conform to all coding recommendations are similar in format and therefore are easier to use and maintain.

You can use modular procedures for general programming or you can group them in procedure libraries. Grouping procedures into libraries is simply a way of collecting procedures so that calling programs can access them easily. When you link your program to a library, the VAX linker will automatically search that library to resolve any references that your program makes to procedures in the library. Because the VAX Linker searches the specified library automatically, your

program can call many modular procedures without having to include the name of each procedure explicitly in the LINK command. The program's executable image and the procedures that it calls are executed in the proper sequence at run time.

Figure 1–2 shows the development of a program that calls one or more procedures in a library. Depending on the options you select when writing modular procedures, you can control the way the linker accesses your procedures and thus the way procedures are invoked at run time. For example, if you place commonly used procedures within a shareable procedure library or shareable image library you can save memory and disk space because all user processes may access a single copy of the shared procedures.

1.3 Existing VAX/VMS System Procedures

There are many system routines included in the VAX/VMS operating system which cater to the needs of users with advanced and varied applications in mind. Before you write a new procedure, you should check to make sure that the application does not already exist.

You may also find that the system procedures provide useful building blocks for your own procedures. Procedures designed to accomplish many general functions already exist in the system libraries and your procedure may simply call an existing procedure rather than duplicate the code.

The following is a list of the four types of callable system procedures along with the titles of the manuals documenting those procedures:

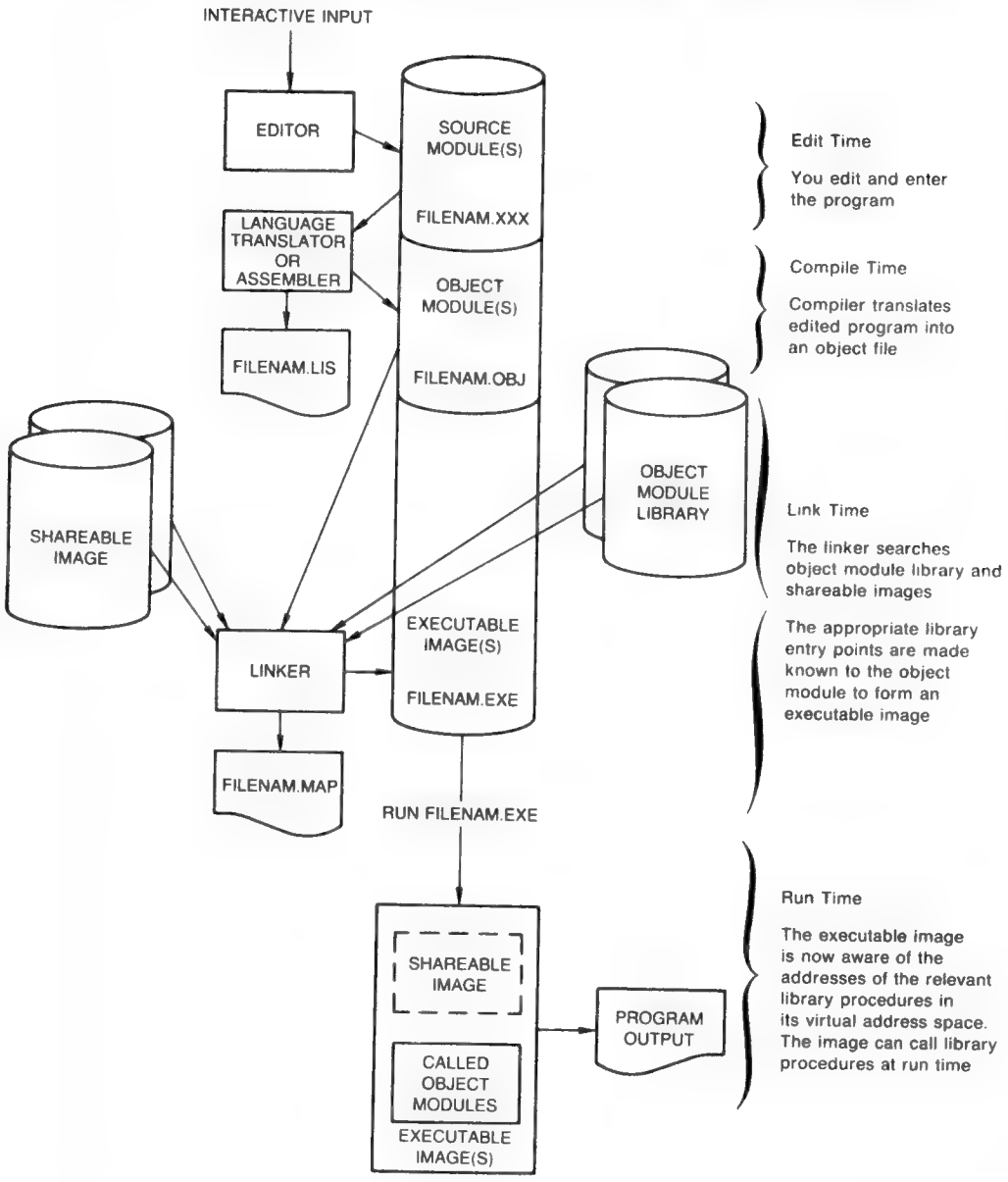
Run-Time Library Procedures
VAX/VMS Run-Time Library Routines Reference Manual

System Services
VAX/VMS System Services Reference Manual

Utility Routines
VAX/VMS Utility Routines Reference Manual

Record Management Services
VAX Record Management Services Reference Manual

Figure 1–2 Developing a Program that Calls Library Procedures



The following sections describe the types of system routines available as part of the VAX/VMS operating system.

1.3.1 Run-Time Library Procedures

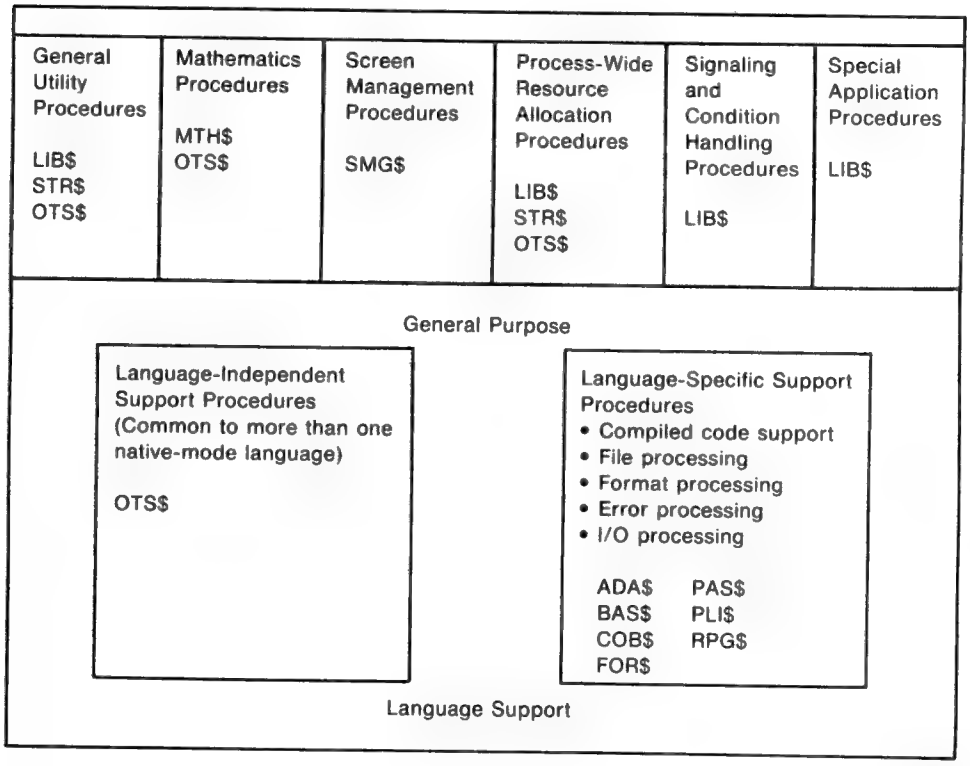
The VAX Common Run-Time Procedure Library (or simply the Run-Time Library) contains two types of procedures:

- General purpose procedures
- Language support procedures

The Run-Time Library provides a common run-time environment for user programs because the Run-Time Library procedures follow the VAX/VMS Modular Programming Standard. For a quick overview of the types of procedures available through the Run-Time Library, see Figure 1–3.

Most general purpose Run-Time Library procedures are fully reentrant and position independent. (Reentrancy is discussed in Section 3.3; position independence is discussed in Section 3.1.1.) Run-Time Library procedures can be used in conjunction with VAX/VMS system services. The advantage of a common run-time environment is that any program written in MACRO, BLISS, or a supported high-level language can call any procedure in the Run-Time Library.

Figure 1–3 Procedures Available in the Run-Time Library



ZK-3082-84

The following chart describes the Run-Time Library procedures.

RTL Procedures	Function
General utility	This group includes procedures for getting a record from a device, performing string manipulation, converting data types for input and output, and obtaining the system date or time.
Mathematics	Mathematics procedures perform common arithmetic, algebraic, and trigonometric functions, such as calculating the sine of an angle.
Resource allocation	Resource allocation procedures allocate and deallocate virtual memory, VAX/VMS local event flag numbers, BASIC/FORTRAN logical unit numbers, and dynamic strings.
Screen management	Screen management procedures perform terminal-independent screen management functions. These procedures assist you in designing, composing, and keeping track of complex images on a video screen.
Signaling and condition handling	These procedures perform operations involved with handling exception conditions such as signaling exceptions, establishing condition handlers, and enabling the detection of hardware exceptions.
Language-independent support	Language-independent support procedures can be called from any VAX language. Although most of these routines are in the OTS$ facility, some LIB$ routines also perform language-support operations. Language-independent support procedures are mostly data type conversion procedures.

RTL Procedures	Function
Special applications	Procedures for specialized applications, such as syntax analysis and cross-reference tasks, are also available in the Run-Time Library.

1.3.2 System Services

System services are procedures that the VAX/VMS operating system uses to control resources available to processes. They are also used to provide for communication among processes; and to perform basic operating system functions such as the coordination of input/output operations.

Although most system services are used primarily by the operating system itself on behalf of logged-in users, many are available for use in application programs. For example, when you log in to the system, the Create Process system service ($CREPRC) is called to create a process on your behalf. You may, in turn, write a program that calls $CREPRC to create a subprocess.

System services can be divided into functional groups. The following list describes the function of each group of system services.

Service Group	Function
Security	The security services provide various mechanisms that you can use to enhance the security of VAX/VMS systems.
Event flag	Event flags can be used by a process to synchronize sequences of operations in a program. Event flag services clear, set, and read event flags, and place a process in a wait state pending the setting of an event flag or flags.

Service Group	Function
AST	Process execution can be interrupted by events (such as I/O completion) to allow for the execution of designated subroutines. These software interrupts are called asynchronous system traps (ASTs) because they occur asynchronously to process execution. There are system services which allow a process to control the handling of ASTs.
Logical names	Logical name services are used to maintain and access character string logical name and equivalence name pairs. Logical names can provide device independence for system and application program input and output operations.
Input/output	I/O services perform input and output operations directly, rather than through the file handling services of the VAX Record Management Services (RMS). I/O services do the following: • Perform logical, physical, and virtual input/output operations • Perform network operations • Queue messages to system processes
Process control	Process control services allow you to create, delete, and control the execution of processes.
Timer and time conversion	Timer services schedule program events for a particular time of the day, or after a specified interval of time has elapsed. The time conversion services provide a way to obtain and format binary time values for use with the timer services.

Service Group	Function
Condition handling	Condition handlers are procedures that receive control when a hardware or software exception condition occurs during image execution. Condition-handling services designate condition handlers for special purposes.
Memory management	Memory management services provide ways to use the virtual address space available to a program. Included are services that do the following: • Allow an image to increase or decrease the amount of virtual memory available • Control the paging and swapping of virtual memory • Create and access files in memory that contain shareable code or data
Change mode	Change mode services alter the access mode of a process. These services are used primarily by the operating system.
Lock management	Lock management services allow cooperating processes to synchronize their access to shared resources.

1.3.3 Utility Routines

Utility routines perform a particular task or set of tasks. For example, the Print Symbiont Modification (PSM) routines can be used to modify the VAX/VMS print symbiont, and the EDT routines can be used to invoke the EDT editor from a program.

When using a set of utility routines that performs the same tasks as a VAX/VMS utility, you should read the documentation for that utility in the *VAX/VMS Utilities Reference Volume*. Doing so will provide you with additional information on the tasks that each set of routines can perform. The following

list shows VAX/VMS utilities that have corresponding utility routines.

Routine Group	Function
Command Language (CLI)	CLI routines process command strings using information from a command table. A command table is a list of definitions that describe the allowable formats for commands.
Convert and Convert/Reclaim (CONV)	CONV routines copy records from one or more files to an output file, changing the record format and file organization to that of the output file. The Convert /Reclaim routine reclaims empty buckets in Prolog 3 indexed files so that new records can be written into them. These routines cannot be called from AST level.
File Definition Language (FDL)	FDL routines perform many of the functions of the RMS File Definition Language. These routines cannot be called from AST level.
Librarian (LBR)	LBR routines are used to create and maintain libraries and library modules. Libraries are files that provide a convenient way to organize frequently used modules of code or text.
Sort/Merge (SOR)	SOR routines are used to implement a sort or merge operation within a program. These routines are reentrant, that is, a number of sort or merge operations can be active at the same time.

The following facilities are accessible only through the utility routines; there are no corresponding VAX/VMS utilities.

Routine Group	Function
Data Compression/Expansion (DCX)	DCX routines are used to analyze and compress data records and then expand the compressed records to their original state. No information is lost in this compression /expansion process.
Editor (EDT)	The EDT routine invokes the EDT editor from within a program. The program may be written to handle the editing work, or EDT may run interactively to allow a user at the terminal to edit a file while the program is running.
Print Symbiont (PSM)	PSM routines modify the behavior of the print symbiont that is supplied with the VAX/VMS operating system.
Symbiont/Job Controller Interface (SMB)	SMB routines provide the interface between the job controller and symbiont processes.

1.3.4 Record Management Services

The Record Management Services (RMS) assist user programs in processing and managing files and their contents. RMS allows you to create files that use a minimum amount of system resources while decreasing input/output time. There are two types of Record Management Services. See the *VAX Record Management Services Reference Manual* for complete descriptions of these Record Management Services.

RMS Service Group	Function
File-related services	File-related services create and access new files, access existing files, extend the disk space allocated to a file, close a file, obtain file characteristics, and perform other functions related to the file as a whole.
Record-related services	Record-related services get, locate, insert, delete, and update records, as well as other record-related operations.

1.4 Tools for Developing Applications on VAX/VMS

There are several tools which are specifically designed to aid you in developing applications in the VAX/VMS environment.

- DEC/CMS

  The Code Management System (CMS) is a program library system for software development and maintenance. CMS works as an online librarian for a project. While project members perform the normal functions of program development, modification and testing, CMS keeps a record of changes made to their files.

- DEC/MMS

  The Module Management System (MMS) is a tool that automates and simplifies the building of software systems. MMS is useful for building both simple programs, which may have only one or two source files, and complex programs, which may consist of several source files, message files, and documentation. It can rebuild all the components in a system, or rebuild only those that have changed since the system was last built.

- DEC/Test Manager

  DEC/Test Manager is a tool that organizes software tests and automates the way you run tests and evaluate the results. It provides an efficient way to organize, run, and store the results of exiting tests. DEC/Test Manager is based on the concept of regression testing.

- VAX Language-Sensitive Editor

  The VAX Language-Sensitive Editor is a source code editor that allows you to quickly and accurately develop programs using the EDT commands. The VAX Language-Sensitive Editor has useful development features such as templates, placeholders, user-defined templates and placeholders, and commands. Other features include a compiler interface, and on line language help.

- VAX PCA

  The VAX Performance and Test Coverage Analyzer (PCA) is a tool that can be used by software developers to gather performance or test coverage data on user programs. This tool will then present that data in tables, bar histograms, annotated source listings, and other formats.

The tools described above are optional products. They are not included with VAX/VMS. For further information on these products, see your DIGITAL sales representative.

2 Design

Well designed procedures are more likely to be modular, well written, and easy to maintain. Any time that is saved by skimping at the design stage is lost several times over in patchwork attempts to remedy problems stemming from a poor design.

This chapter will help you with the following aspects of designing your new application:

- Organizing new applications
- Designing a modular procedure interface
- Using system resources
- Using I/O
- Beginning the internal documentation
- Planning for signaling and condition handling

2.1 Organizing New Applications

The first work to be done in designing a new application is looking at the overall organization. Your application will be constructed of one or more files which will each contain one or more procedures. When linked, your procedures will be organized into program sections (PSECTs). Each procedure, as well as the interface between the procedures, must be designed to conform to the VAX/VMS Modular Programming Standard (Appendix A).

2.1.1 Organizing Files and Modules

Within your application, you will have one or more files. Each file will contain exactly one module. For information on naming files, refer to Section 3.1.2.3. For information on naming modules, refer to Section 3.1.2.4.

2.1.2 Organizing Procedures into Modules

Each module should contain a single procedure or a group of related procedures. The VAX/VMS Linker always brings the entire module containing a called procedure into the image if any of its entry points are referenced. Thus, placing each procedure in a separate module reduces image size. It also allows more flexibility when using a procedure library because you can supply your own version of one procedure while using other procedures from the library. If many procedures have been grouped in a single module the linker must link all of them or none.

You should group procedures into a module only if they share the same static storage; or have a similar calling sequence, perform similar functions, and share a significant amount of code.

If you are writing a large number of related procedures that call one another or access common data blocks, you should make the relationship among those procedures as clear as possible. You can do this by organizing those procedures to minimize the interaction between procedures, and between procedures and data structures. There are several guidelines to help you minimize such interaction:

- Organize procedures into levels of abstraction.
- Make sure each level calls only the next lower level.
- Restrict read/write access to data structures and system components to as few procedures as possible.

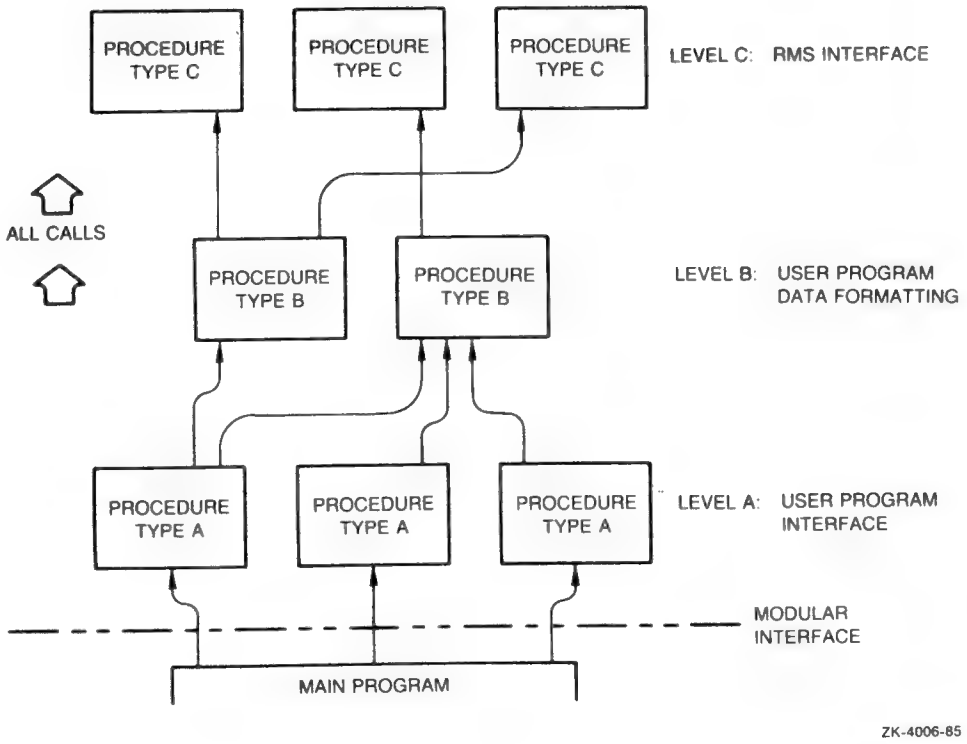
Figure 2–1 shows the BASIC and FORTRAN record I/O processing procedures. These are implemented in the following three levels of abstraction:

1 User program interface (UPI)

2 User program data formatting (UDF)

3 Record processing and VAX RMS interface (REC)

Figure 2–1 Levels of Abstraction



All calls are made in one direction, to the next innermost level. Procedures at different levels should be in different modules. Figure 2–2 shows possible groupings of procedures.

2.2 Defining a Modular Procedure Interface

Procedures communicate with one another by passing arguments. In order to clarify the interactions between procedures and programs, each argument must be defined in the design stage.

2.2.1 Explicit Arguments

Explicit arguments are a procedure's primary interface with other programs. Therefore, rules for argument order, data types, and passing mechanisms must be followed carefully to maintain a modular interface. The following format is used to describe each argument.

```
argument-name
VMS usage:    argument-data-structure
type:         argument-data-type
access:       argument-access
mechanism:    argument-passing-mechanism
```

(See Appendix B for descriptions of each of these four argument attributes.)

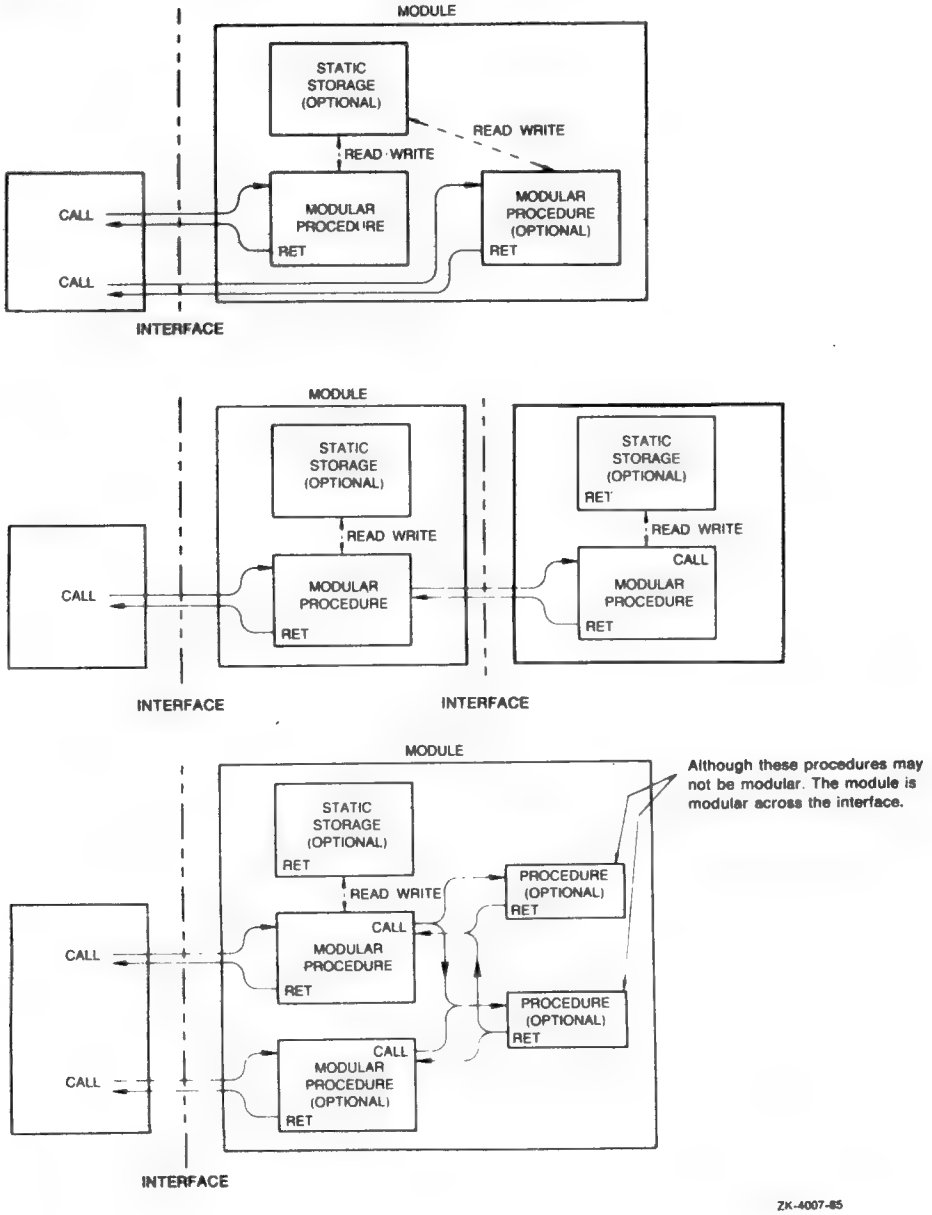
To make your procedures easier to call the passing mechanism used for particular data types should be consistent throughout all procedures in a facility. It is recommended that you pass all atomic data by reference and all string data by descriptor.

2.2.2 Implicit Arguments

An implicit argument is an argument that is not specified in the argument list. Implicit arguments provide additional information to your procedure from static storage locations. There are two types of implicit arguments:

- Those allocated by the calling program
- Those allocated by your procedure

Figure 2–2 Possible Procedure Groupings



The use of implicit arguments is discouraged because they violate the VAX/VMS Modular Programming Standard (Appendix A). If you do not use implicit arguments, modularity is easier to maintain. If your procedure must retain information from previous activations read Section 2.2.3 for ways to avoid using implicit arguments.

2.2.2.1 Implicit Arguments Allocated by the Calling Program

The calling program can allocate implicit arguments as statically allocated variables in a named PSECT (for example, COMMON and MAP in BASIC, COMMON in FORTRAN, or variables declared in the outer block of a procedure or program in PASCAL). The calling program can also allocate implicit arguments as statically allocated global variables (for example, symbols defined with a double colon [::] in MACRO, and GLOBAL variables in BLISS).

Allocation of implicit arguments by the calling program violates the VAX/VMS Modular Programming Standard for the following reasons:

- Two programs could use the same PSECT name or global variable for different values. This error will be undetected.
- The calling program is no longer independent of the called procedure. Consequently, a change in one could inadvertently affect the other.
- In FORTRAN, the calling program has to declare all variables as COMMON regardless of the number of implicit inputs actually needed. All COMMON variables should also be declared by all modules that use the COMMON storage, further decreasing independence.

2.2.2.2 Implicit Arguments Allocated by the Called Procedure

Implicit arguments allocated by the called procedure are kept in local static storage.

These implicit arguments are usually used to keep track of resources (using resource allocating procedures) and shorten the explicit argument list. However, the use of implicit inputs by non-resource-allocating procedures can lead to unexpected results. Assume that procedure A is to leave information for a companion procedure B. Thus, B has both explicit inputs (from its caller) and implicit inputs (from A's storage). Next, consider that a calling program calls A, then calls procedure X, and finally calls B. For the calling program to get correct results from B, it must know that X (and any procedure that X calls) did not make a call to A (since such a call would change the implicit inputs A leaves for B).

One of the objectives of modular programming is to permit procedures to be combined arbitrarily without needing to understand each other's internal workings. Therefore, the use of such implicit arguments violates the VAX/VMS Modular Programming Standard. The same problems can also occur with any non-resource-allocating procedure that leaves results for itself as future implicit arguments.

2.2.3 How to Avoid Using Implicit Arguments

There are three ways to write procedures that do not allocate resources to avoid the implicit argument problems described in Section 2.2.2.

- When one procedure obtains results from another, combine the two procedures into a single call. (See Section 2.2.3.1.)
- Provide a single call to an action routine that is supplied by the calling program part way through your procedure's execution. (See Section 2.2.3.2.)
- Give the calling program responsibility for retaining information from a procedure activation. This is done with an explicit argument. (See Section 2.2.3.3.)

2.2.3.1 Combining Procedures

Often non-resource-allocating procedures can be combined into a single procedure that returns all information explicitly in a single call.

To see the effects of combining procedures to avoid the use of implicit arguments, compare Example 2–1 with Example 2–2.

Example 2–1 FORTRAN Program Showing the Improper Use of Implicit Arguments

```
!+
! This program demonstrates a situation where
! the input of a procedure depends on the output
! of a previously called procedure.
!-
        REAL*4 X, Y, RESULT
        X = 1
        Y = 1
!+
! Call the procedure that writes into a common data area.
!-
        CALL SUM_SQUARES (X, Y)
!+
! Call the procedure that reads from the common data area.
!-
        CALL GET_SQRT (RESULT)
!+
! Print the result obtained.
!-
        WRITE (6,10) X, Y, RESULT
10      FORMAT(1X, 'SQRT(', F6.2, '**2 + ', F6.2, '**2) =',F6.2)
        STOP
        END

!+
! This procedure sums the squares of its two inputs and
! places the result in a common area, for use by some
! other procedure.
!-
        SUBROUTINE SUM_SQUARES (A, B)
        COMMON /INTERNAL_STORAGE/ TEMP_RESULT
        TEMP_RESULT = (A ** 2) + (B ** 2)
        RETURN
        END
!+
! This procedure calculates the square root of whatever
! number is in the common area.
!-
        SUBROUTINE GET_SQRT (C)
        COMMON /INTERNAL_STORAGE/ TEMP_RESULT
        C = SQRT (TEMP_RESULT)
        RETURN
        END
```

Example 2–2 FORTRAN Program Combining Procedures to Avoid Implicit Arguments

```
!+
! This procedure shows the subroutines called in
! the previous example combined into a single subroutine
! that eliminates the use of COMMON.
!-
        REAL*4 X, Y, RESULT
        X = 1
        Y = 1
!+
! Call the new procedure.
!-
        CALL DO_IT_ALL (X, Y, RESULT)
        WRITE (6,10) X, Y, RESULT
10      FORMAT (1X, 'SQRT (', F6.2, '**2 + ', F6.2, '**2) = ',F6.2)
        STOP
        END

!+
! This procedure calculates the square root of the sum of
! the squares of its first two arguments, and returns the
! result in the third argument.  It combines the functions
! provided by the SUM_SQUARES and GET_SQRT
! procedures and eliminates the use of COMMON.
!-
        SUBROUTINE DO_IT_ALL (A, B, C)
        C = SQRT ((A ** 2) + (B ** 2))
        RETURN
        END
```

2.2.3.2 User Action Routine

Another way to combine several procedures into one call is to let the calling program gain control at a critical point in your procedure's execution. For this to happen, your procedure must specify an action routine argument that will be called during execution. Thus, your procedure can execute twice, before and after the action routine, with no implicit inputs. The OPEN statements in BASIC FORTRAN and PASCAL use this technique by permitting the user to supply a user action routine.

To keep the calling program from having to provide implicit inputs for its action routine, your procedure should also provide another argument that is passed to the action routine. The

calling program uses the following calling sequence to invoke your procedure:

```
CALL my-proc (... ,action-routine ,user-arg)
```

Then your procedure invokes the action routine as follows:

```
CALL action-routine (... ,user-arg)
```

For information on writing user-action routines, see Section 3.1.6.

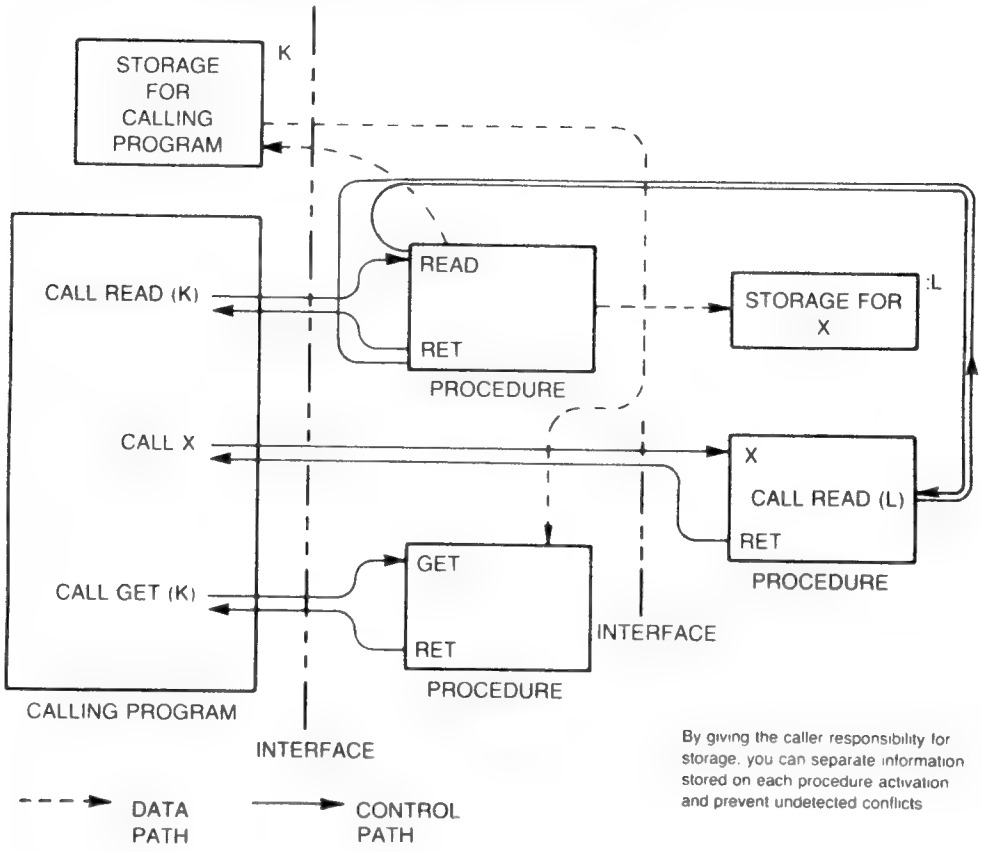
2.2.3.3 Designating Responsibility to the Calling Program

You can make the calling program responsible for retaining information from one procedure activation to another. Here are three ways to accomplish this:

- Require the calling program to allocate the storage your procedure needs. Then have the calling program pass the address of the storage location as an explicit argument on all calls to your procedure. The disadvantage of this method is that you cannot increase the amount of storage needed by your procedure without requiring all calling programs to be rewritten. Thus, you should use this method only when you are confident that your procedure will not be revised to use additional storage in the future.
- Require the calling program to allocate a longword pointer to the stored data and pass its address to your procedure as an explicit argument. On the first call, your called procedure will dynamically allocate storage (by calling LIB$GET_VM) and store its address in the caller's longword. On subsequent calls, your procedure will use information left in the storage area from previous calls.
- Require the calling program to pass a processwide identifying value to your procedure on all calls. The processwide identifier indicates which information from previous procedure activations is to be used as implicit inputs.

Figure 2–3 shows a calling program with responsibility for explicitly indicating the storage to be used by the called procedure.

Figure 2–3 Designating Storage Responsibility to the Caller



calling program uses the following calling sequence to invoke your procedure:

```
CALL my-proc (... ,action-routine ,user-arg)
```

Then your procedure invokes the action routine as follows:

```
CALL action-routine (... ,user-arg)
```

For information on writing user-action routines, see Section 3.1.6.

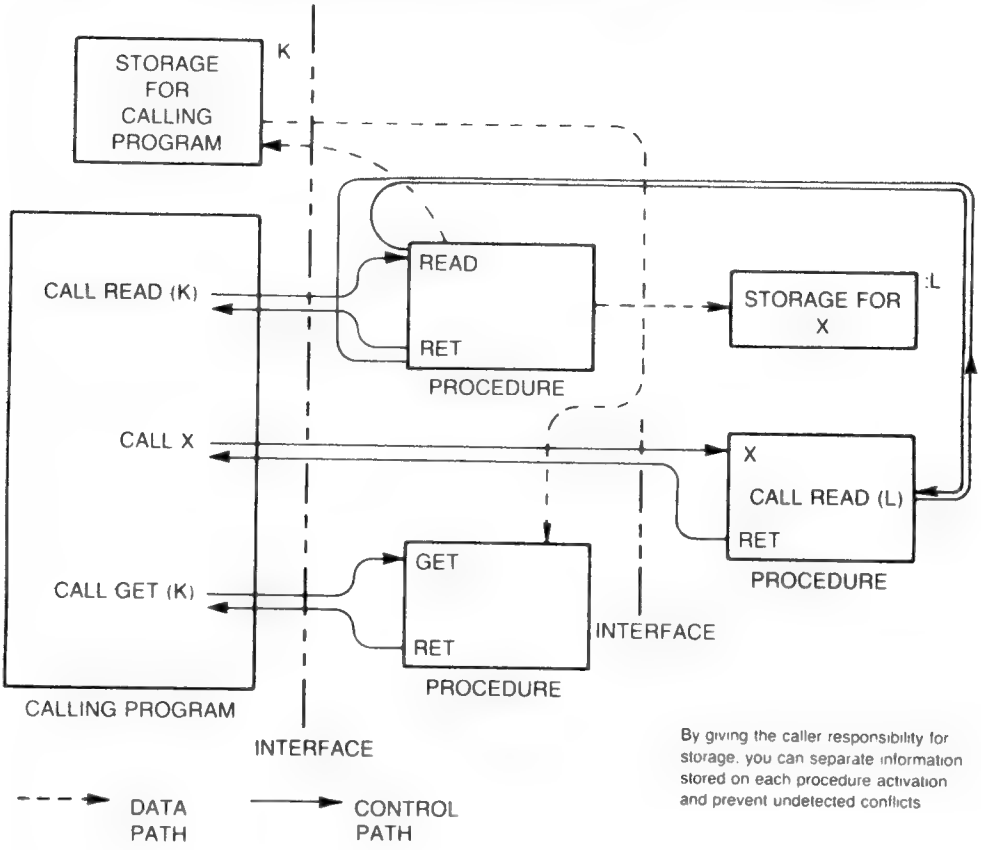
2.2.3.3 Designating Responsibility to the Calling Program

You can make the calling program responsible for retaining information from one procedure activation to another. Here are three ways to accomplish this:

- Require the calling program to allocate the storage your procedure needs. Then have the calling program pass the address of the storage location as an explicit argument on all calls to your procedure. The disadvantage of this method is that you cannot increase the amount of storage needed by your procedure without requiring all calling programs to be rewritten. Thus, you should use this method only when you are confident that your procedure will not be revised to use additional storage in the future.
- Require the calling program to allocate a longword pointer to the stored data and pass its address to your procedure as an explicit argument. On the first call, your called procedure will dynamically allocate storage (by calling LIB$GET_VM) and store its address in the caller's longword. On subsequent calls, your procedure will use information left in the storage area from previous calls.
- Require the calling program to pass a processwide identifying value to your procedure on all calls. The processwide identifier indicates which information from previous procedure activations is to be used as implicit inputs.

Figure 2–3 shows a calling program with responsibility for explicitly indicating the storage to be used by the called procedure.

Figure 2–3 Designating Storage Responsibility to the Caller



ZK-4004-85

Calling Program Allocates Procedure Storage

This method casues the calling program to allocate all storage needed and pass the address of the storage as an explicit argument on each call.

For example, the library procedure MTH$RANDOM requires that the calling program allocate storage for the longword seed and pass its address on each call. MTH$RANDOM takes the seed as input and computes the next random number sequence from the current seed value. MTH$RANDOM returns a random number between 0 and 1. It also updates the longword seed passed by the calling program. This ensures that the procedure will generate a different value on the next call.

The next two sections describe interface techniques that permit storage size to change without affecting the interface with the calling program.

Calling Program Passes Pointer

In this method, the calling program allocates only a longword pointer to the dynamic heap storage to be allocated by your procedure. It then passes the address of the longword as an explicit argument. There are two interface techniques that can be used to indicate storage is to be initialized. Following is a description of each of these techniques:

- Provide a single entry point. If your called procedure finds the value zero in the longword that the calling program has allocated, the procedure allocates and initializes dynamic heap storage.
- Provide a second entry point. This entry point stores the address of the allocated storage in the longword. On subsequent calls, your procedure uses that value as the storage address of information from previous calls.

Regardless of the method used to indicate storage allocation and initialization, you must also provide a way to indicate storage deallocation. You can do this by using either a separate argument or separate entry point.

For example, the procedure LIB$INIT_TIMER, which gets times and counts from the operating system, uses a single optional argument **handle-adr** to determine where these values are to be stored. The **handle-adr** argument is the address of a longword

pointing to a block of storage that contains the values of times and counts.

- If **handle-adr** is missing, the values are stored in static storage, making this call non-AST-reentrant.
- If **handle-adr** is zero, LIB$INIT_TIMER allocates a block of dynamic heap storage by calling LIB$GET_VM. The values are placed in that block, and the address of the block is returned in **handle-adr**.
- If **handle-adr** is nonzero, it is considered to be the address of a storage block previously allocated by a call to LIB$INIT_TIMER. The block will then be used again, and new times and counts are stored in it.

LIB$FREE_TIMER deallocates the block of dynamic heap storage allocated by a previous call to LIB$INIT_TIMER. The **handle-adr** argument to LIB$FREE_TIMER is the address of a longword that points to a block of dynamic heap storage where times and counts have been stored. That storage is returned to free storage by calling LIB$FREE_VM.

Calling Program Passes a Processwide Identifier

In this method, the calling program passes a processwide identifying value to identify implicit results produced on previous calls which will be implicit inputs on this call. Any calling program can use the processwide identifier. Examples of processwide identifiers include BASIC or FORTRAN logical unit numbers and VAX/VMS system services I/O channel numbers.

Processwide identifiers are a resource. Modular programming techniques require that all resources allocated by a procedure be allocated by calling a resource-allocating procedure. This prevents conflicts because a single procedure can keep track of multiple allocations to more than one procedure or procedure activation. Therefore, if you use the method described in this section, you will also have to write a resource-allocating procedure to control the resource. If you write a resource-allocating procedure, it is recommended that you place it in an object module library so that other programmers can use it.

The library procedures LIB$GET_LUN and LIB$FREE_LUN allocate and deallocate FORTRAN and BASIC logical unit numbers outside the range normally specified in user programs,

that is, outside the range 0 to 99. An example of a resource-allocating procedure that allocates identifying numbers is given in Section 2.3.2.

2.2.4 Order of Arguments

Procedures in the Run-Time Library follow a consistent pattern for positioning arguments. You should follow the same guidelines. Group procedure arguments from left to right in this order:

1. Required input arguments (read access)
2. Required input-output arguments (modify access)
3. Required output arguments (write access)
4. Optional input arguments (read access)
5. Optional input-output arguments (modify access)
6. Optional output arguments (write access)

Note that optional arguments follow required arguments. Therefore, when the calling program omits the optional arguments, the actual argument list passed to the procedure is shortened.

The called procedure accesses the required arguments from left to right, beginning with the first argument. The only exceptions are procedures that return a function value longer than 64 bits, such as strings or H_floating values. Most function values are returned in the first two registers, R0/R1. (This is done automatically by high-level languages.) However, when the function value exceeds 64 bits, it is too large to be returned in R0/R1. In this case, the calling program uses the first argument to specify where the function value is to be stored, and the other arguments are shifted right one position. (See the VAX Procedure Calling and Condition Handling Standard in the *Introduction to VAX/VMS System Routines*.)

2.2.5 Using Optional Arguments

An optional argument is one that the calling program can omit. The calling program indicates the omission by passing argument list entries containing zero. For a trailing optional argument, the calling program can pass a shortened list or a zero argument list entry.

A zero argument list entry is simply a zero passed to the procedure by value. For example, if we call a procedure called GRA_CUBE and omit an optional argument C, the calling sequence from BASIC would be as follows:

```
15 CALL GRA_CUBE(A, B, 0 BY VALUE)
```

In this call, "0 BY VALUE" is the zero argument list entry.

Note: Most VAX/VMS system services, unlike the Run-Time Library procedures, cannot accept a shortened argument list. Omitted arguments must always be indicated with a zero argument list entry.

For arguments passed by value, there is no distinction between passing a zero value and passing a zero argument list entry.

2.3 JSB Entry Points

DIGITAL recommends that you do not use JSB entry points in procedures which will be contained in a procedure library. Procedures which can be invoked only by JSB instructions are not callable by high-level languages and violate the VAX Procedure Calling and Condition Handling Standard. If a procedure does use a JSB entry point, it must also provide an equivalent call entry point to maintain language independence. The call entry point must be provided because JSB instructions are only available in MACRO and BLISS32.

If you provide a JSB entry point for your procedure, the name of the JSB entry point is the same as the name of the procedure, except that it ends in _Rn. The n indicates the highest register modified or used as an input argument.

For example, the JSB entry point of the Run-Time Library procedure LIB$ANALYZE_SDESC is LIB$ANALYZE_SDESC_R2.

2.4 Using System Resources

The system resources available to you are limited by your account quotas and by the amount of available resources on the system. Efficient use of system resources makes more resources available for all processes.

2.4.1 Choosing a Storage Type

There are three types of storage: static, stack, and heap. The three forms of storage differ in the method and duration of allocation, that is, how long that storage is in use.

2.4.1.1 Stack Storage

A procedure allocates stack storage on the process stack at run time. The procedure allocates stack storage dynamically, as it is needed. To allocate stack storage, the procedure moves the stack pointer "up" by decreasing its value. The procedure deallocates stack storage by moving the stack pointer "down" (increasing its value) when that procedure returns control to the calling program. Stack storage exists only for the duration of the procedure activation that creates it.

2.4.1.2 Heap Storage

Dynamic heap storage is allocated at run time from a processwide pool, as the procedure activation needs it and as the account quotas and virtual address space of your process permits.

To allocate heap storage, your procedure calls a system routine such as the Run-Time Library procedure LIB$GET_VM or the system service $EXPREG. The call to the system routine may be within the procedure itself, or you may use a general resource-allocating procedure to centralize your resource allocations.

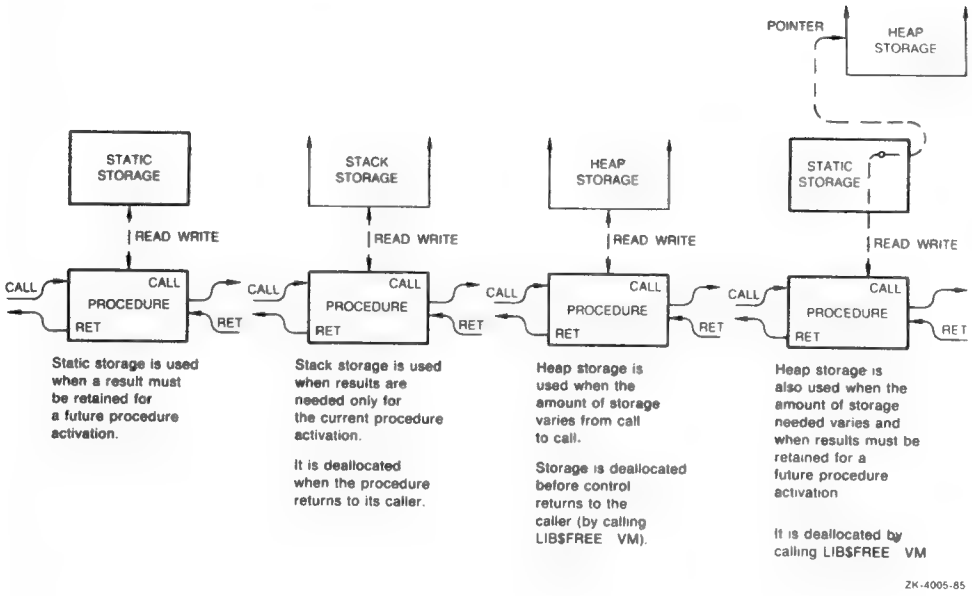
Heap storage is deallocated—that is, returned to the processwide pool—by calling LIB$FREE_VM. The system service $CNTREG cannot be used to deallocate heap storage.

Figure 2–4 shows how the different types of storage are used.

Note: The type of storage to be used can be determined by the duration and/or quantity of the storage. Any storage that

is of long duration and unknown quantity (at compile time) should be heap storage. Storage of short duration (during the current invocation of the procedure) should be stack storage. Storage of long duration that is needed in only one instance should be static storage.

Figure 2–4 Use of Storage Types



2.4.1.3 Static Storage

At link time, the VAX/VMS Linker will collect storage in similar PSECTs into a single image section. The initial contents of this storage are specified in the source program. The linker will initialize any non-initialized static storage to zero. On calls to a procedure after initialization, the static storage has the same allocation and the contents left from the previous call.

2.4.1.4 Avoiding Use of Static Storage

There are several disadvantages to using static storage:

- Static storage is an inefficient use of memory. When using static storage, you must provide for the largest possible memory use.
- Because of the inefficient use of memory, your image size will be larger.
- Use of static storage can easily lead to problems with AST reentrancy, as seen in Example 2–3. This example circumvents the problem of of an AST corrupting data by setting a first-time flag. Another method of preventing this problem is to use "test and set" instructions. For further information, see Section 3.3.4.2.

Example 2–3 Static Storage and AST Reentrancy

```
10      !+
        ! Program to demonstrate corruption
        ! of static storage due to AST's.
        !-
        DECLARE LONG CURRENT_NUMBER
        !+
        ! Enable CTRL/C AST handling.
        !-
        ON ERROR GOTO 19000
        X% = CTRLC
        !+
        ! Increment the number and print the
        ! current value.  When the number
        ! reaches 1000, exit.
        !-
        FOR CURRENT_NUMBER = 1% TO 1000%
100     PRINT CURRENT_NUMBER;
        NEXT CURRENT_NUMBER
        GOTO 32767
19000   !+
        ! Error-handling routine.  If this routine is
        ! entered due to a CTRL/C
        ! AST, corrupt CURRENT_NUMBER by setting it to -1.
        !-
        IF ERR = 28 THEN CURRENT_NUMBER = -1%
        RESUME 100
32767   END
```

2.4.1.5 Summary of Storage Use by Language

Table 2–1 summarizes storage available to the programmer in various language procedures.

Table 2–1 Summary of Storage Use by Language

Language	Storage Type		
	Static	**Stack**	**Heap**
Ada	Constants and fixed-size objects contained in library packages	Local subprogram and task variables	Dynamically sized objects in library packages and objects created by allocators
BASIC	All COMMON and MAP data storage	Local variables	Dynamic strings
		Most arrays	Executable DIMENSION statement
BLISS	OWN and GLOBAL	STACK LOCAL	By calling LIB$GET_VM
COBOL	All data storage	Not applicable	By calling LIB$GET_VM
FORTRAN	All data storage	Not applicable	By calling LIB$GET_VM
MACRO	Block storage	Decrementing stack pointer	By calling LIB$GET_VM
PASCAL	All program or module level storage	PROCEDURE and FUNCTION local	By calling NEW*
PL/I	STATIC	AUTOMATIC	ALLOCATE statement (BASED)**
RPG II	All data storage	Not applicable	By calling LIB$GET_VM

Note: * Although this is true most of the time, there are other rules which can also determine STATIC versus STACK allocation. For further information, see Section 2.2.1 in the VAX PASCAL User's Guide.

**** BASED is the storage class used to allocate heap storage in PL/I. The ALLOCATE statement does the actual allocation.**

2.4.2 Using Event Flags

Event flags allow modular procedures to communicate with each other and synchronize their operations. Because they can be allocated at run time, event flags allow your procedure to run independently of other procedures existing in the same process.

Event flags are allocated and deallocated by the Run-Time Library procedures LIB$GET_EF and LIB$FREE_EF. (For further information, see the *VAX/VMS Run-Time Library Routines Reference Manual* and the descriptions of the LIB$GET_EF and LIB$FREE_EF procedures which appear in Part II of that manual.)

2.4.3 Using Logical Unit Numbers

A logical unit number is used to define the device or file a program uses to perform input and output. Modular procedures do not need to know the unit numbers of other procedures running at the same time.

Logical unit numbers are used only in BASIC and FORTRAN.

Logical unit numbers should be allocated and deallocated using the LIB$GET_LUN and LIB$FREE_LUN Run-Time Library procedures. (For further information on using logical unit numbers, see the descriptions of the LIB$GET_LUN and LIB$FREE_LUN procedures in Part II of the *VAX/VMS Run-Time Library Reference Manual.*

2.5 Using Input/Output

In general, your procedure's input/output will be directed to either the terminal screen or a file. (In some cases, you may need to use mailbox I/O and network operations. For information on these areas, see *Guide to Networking on VAX/VMS.*) Regardless of whether you are concerned about input/output to the terminal screen or to a file, there are two rules which must be followed to maintain modularity:

- A procedure must not print error or informational messages either directly or by calling the $PUTMSG system service. It must either return a condition value in R0 as a function

value, or call LIB$SIGNAL or LIB$STOP to output all messages. (LIB$SIGNAL and LIB$STOP may be called either directly or indirectly.)

- A procedure should use device-independent services and procedures for input/output.

2.5.1 Terminal Input/Output

The methods available for performing input/output (I/O) to the terminal include the following:

- Queue I/O Request system service ($QIO)

  Using a $QIO to perform terminal I/O is very efficient, however, it is also the most difficult method to use. $QIOs use device-dependent services and are the most difficult to use from high-level languages of all methods discussed here, because there are more steps involved, and the calling interface requires more knowledge from the caller than RMS services. There are several steps which may be required for your procedure to use a $QIO. These may include constructing item lists, writing AST routines, assigning an I/O channel, queuing an I/O request, testing to ensure that the request was successfully queued and completed, and deassigning the I/O channel. You may wish to use a $QIO to perform something that simply can't be done from RMS. (For further information on $QIOs, see Section 7 of the *VAX/VMS System Services Reference Manual*.)

- Record Management Services (RMS)

  RMS services are device independent and provide general purpose services which are easier to call than $QIOs. However, it is often not convenient to construct the access control blocks (FAB, RAB, etc.) required by RMS services from a high-level language. (For further information on RMS services, see the *VAX Record Management Services Reference Manual*.)

- Language I/O statements

  Language I/O statements are provided for all high-level languages. These statements are easy to use and provide simple I/O and data formatting for the high-level language user in the easiest method possible. Native language I/O statements make it unnecessary for the high-level language user to call $QIO or RMS services directly; these calls are

made by the compiled code on your behalf. However, low-level and medium-level languages (VAX MACRO and BLISS32) have no built in language I/O statements and must use $QIO and RMS services for terminal and file I/O. (For further information, see your appropriate language reference manual.)

- Screen Management Procedures in the Run-Time Library (SMG$)

  SMG$ procedures provide an easy to call interface for high-level languages. They are device independent and aid in the composition of complex screen images. The SMG$ facility in the Run-Time Library provides "windowing"; that is, SMG$ makes it easy for an application to divide its screen into multiple regions and provides functions for manipulating those regions. Other features provided by SMG$ procedures are:

  1. Output to virtual displays
  2. Input from a virtual keyboard
  3. The ability to perform asynchronous input
  4. Built-in minimal screen updating
  5. Optional buffering and batching to optimize performance
  6. The ability to trap broadcast messages
  7. The option of performing output to a file or a hardcopy device
  8. Support for foreign (non-DIGITAL) terminals

  For further information about SMG$ procedures, see Section 3 of the *VAX/VMS Run-Time Library Routines Reference Manual*.

During I/O to the terminal it is important that the procedure and the main program cooperate in controlling the terminal screen. For example, an I/O procedure may write something to the terminal screen that the calling program wants to erase. The calling program must know both what and where that information is, in order to erase it. The calling program and the called procedure must communicate by passing arguments that define which part of the screen will be accessed by each.

The Run-Time Library contains Screen Management (SMG$) procedures for this purpose.

It is essential that you do not combine different methods of I/O within your application. Problems may arise if the calling program and the called procedure use different methods of I/O. Each method of performing input/output maintains some knowledge of what is on the terminal screen. At the very least, the current cursor position is remembered. If another type of I/O is performed, that information is not updated and therefore becomes incorrect. The results of any subsequent I/O would be unpredictable.

2.5.2 File Input/Output

File I/O can be performed by the following methods:

- Record Management Services (RMS)

  RMS provides a variety of file organizations and read access modes from which you can select the processing techniques best suited to your application. RMS supports the sequential, relative, and indexed-sequential file organizations. These modes allow you to access records within these files sequentially, randomly by key value or relative record number, or randomly by the record's file address (RFA). It is usually not necessary to call RMS services directly from high-level languages. Consult your language reference manual for specific information on performing record management operations in your language. (For further information on RMS services, see the *VAX Record Management Services Reference Manual.*)

- Language I/O

  The compiled code in most high-level languages will call a Run-Time Library language support procedure for file operations. The Run-Time Library procedures normally call RMS services. Thus, most RMS features are available to the high-level languages user without calling RMS directly. Language I/O statements are suitable for either data files or output files. Low- and medium-level languages (VAX MACRO and BLISS32) do not have any language I/O statements and must call RMS services directly. (For further information, see the appropriate language reference manual.)

2.6 Beginning the Internal Documentation

You must document each procedure carefully so you and others know what the procedure does. In most cases, a module should contain only one procedure.

2.6.1 How to Write a Module Description

You should place a preface containing the following information at the front of each module:

Title: Gives the module name followed by a one-line functional description.

Version: Gives the version and a three-digit edit number. Generally 1-001 is the original version.

Facility: Gives a description of the library facility, such as general utility library (LIB).

Abstract: Gives a short (three to six lines) functional description of the module.

Environment: Lists any special environmental assumptions that the module can make. These include assumptions made at both compilation and execution time that could affect either the hardware or software environments.

Describe situations that the module assumes during execution time, and describe the optional elements of the VAX/VMS Modular Programming Standard that your module does not follow.

Indicate the reentrancy characteristics of the procedures in this module. Each procedure will be either fully-reentrant, AST-reentrant, or non-reentrant.

Author: Include your name and the date you created the module.

Modified by: Include the modification number, name of modifying programmer, modification date, and a list of the modifications.

This concludes the preface. End the preface with a page delimiter. The Actual code for the module follows the preface.

Figure 2–5 shows a sample module description.

Figure 2–5 A Sample Module Description

```
        PROGRAM GRA_CUBE                ! Create representation of a cube
!+
! VERSION:       1-002
!
! FACILITY:      User Graphics Computation Library
!
! ABSTRACT:      This module contains a procedure to create a mathematical
!                representation of a cube, GRA_CUBE.
!
! ENVIRONMENT:   User Mode, AST-reentrant
!
! AUTHOR:        Denise Weldon-Siviy     CREATION DATE:  14-Sep-1984
!
! MODIFIED BY:
! 1-001 - Original.  DWS 14-Sep-1984
! 1-002 - Fix a minor bug in cube volume computation.  MDL 15-Mar-1985
!-
```

2.6.2 How to Write a Procedure Description

You should place a procedure description at the beginning of each procedure in a module.

Always list each of the following topics regardless of whether or not they are actually present. For example, if a procedure has no implicit inputs, write the following:

```
!
!  Implicit Inputs:
!
!       NONE
!
```

The procedure description includes the following elements:

Functional description:

The functional description describes a procedure's purpose and completely documents its interfaces.

The description should include the basis for any critical algorithms used, including literature references where applicable. It should also explain why a particular algorithm was chosen.

Indicate the reentrancy characteristics of this procedure if they differ from those given in the module description.

Calling sequence:

A procedure's calling sequence should include these things in the following order:

1. a return status, value argument, or CALL instruction
2. the procedure name
3. the argument list (typically a list of registers or arguments)

In MACRO, each argument is symbolically defined as the offset relative to the argument pointer (AP).

List the arguments in the order they will appear in a high-level language. Each argument characteristic should also be included, using the procedure argument notation described in Appendix B.

Formal arguments:

List any explicit input, input-output, or output arguments. Include a qualifying description with each argument. You should list the arguments in the order they are listed in the calling sequence.

Implicit inputs:

List any inputs from storage, internal or external to the module, that are not specified in the argument list. Usually all that will appear here is "NONE". See Section 2.2.2.

Implicit outputs:	List any outputs to internal or external storage that are not specified in the argument list.
Completion status or routine value:	List the success or failure condition value symbols that could be returned as completion codes in R0. If your procedure returns a function value other than a condition value in R0, change the heading to "Routine value".
Side Effects:	Describe any functional side effects not evident from a procedure's calling sequence. This includes changes in storage allocation, process status, file operations, and possible signaled conditions. In general, you should document anything out of the ordinary that the procedure does to the environment. If a side effect modifies local or global storage locations, document it in the implicit output description instead.

Figure 2–6 shows a sample procedure description.

Figure 2–6 A Sample Procedure Description

```
!++
! FUNCTIONAL DESCRIPTION:
!
!       Return the system date and time, using the caller's
!       semantics for his/her string.
!
!       Non-reentrant; uses static storage.
!
! FORMAL ARGUMENT(S):
!
!       RESULT_ADDR
!       VMS USAGE : char_string
!       TYPE      : character string
!       ACCESS    : write only
!       MECHANISM : by descriptor
!
!      Address of the descriptor into which the
!      system date and time is written.
!
! IMPLICIT INPUTS:
!
!       NONE
!
! IMPLICIT OUTPUTS:
!
!       NONE
!
! COMPLETION CODES:
!
!       SS$_NORMAL      Procedure successfully completed
!       LIB$_STRTRU     Success, but source string truncated
!
! SIDE EFFECTS:
!
!       Requests the current date and time from VAX/VMS.
!
!--
```

2.7 Planning for Signaling and Condition Handling

There are two methods available to a procedure to indicate to its caller whether it completed successfully. One method is to return a condtion value in R0. The other method is to signal an error condition.

To provide a better user interface, all procedures in a facility should either return condition values or signal error conditions. Regardless of which method you choose, you should be consistent within the facility to make the procedures easier for the user to call.

2.7.1 Guidelines for Signaling Error Conditions

The signaling of an error condition is, in some instances, mandatory. Procedures which return a function value in R0 can not return a condition value and therefore must signal any error conditions encountered.

Other procedures may wish to signal error conditions to maintain efficiency. Checking the return status of a called procedure for repetitive calls can be time consuming and adversely affect the performance of the calling program. For example, if you are going to call a procedure 100 times within a loop and the chances of that procedure's failure are relatively small, you may not want to take the time to check the return status after each call to make sure that the condition value returned was SS$_NORMAL. Signaling error conditions is far more efficient in this type of application.

From the point of view of the calling program, handling a signaled condition is slightly more difficult than checking a returned condition value. It is more complicated because it involves writing a condition handler to be invoked in the event that an error condition is signaled. However, handling a signaled condition allows the calling program to execute more efficiently.

To signal an error condition your procedure uses either a condition handling mechanism provided by the source language, or it calls the Run-Time Library procedure LIB$SIGNAL. To use LIB$SIGNAL, your procedure calls LIB$SIGNAL and specifies the condition code and zero or more arguments that specify the environment of the condition. For

further information about using LIB$SIGNAL, see Part II of the *VAX/VMS Run-Time Library Routines Reference Manual*. The Run-Time Library also contains other procedures to help you with signaling and condition handling. These are described in Section 7, Table 7-1 of the *VAX/VMS Run-Time Library Routines Reference Manual*.

2.7.2 Guidelines for Returning Condition Values

From the point of view of the calling program, it is considerably easier to check returned condition values than to handle signaled error conditions. When the condition value is being returned, the calling program does not need to include a condition handler. The calling program needs only to check the status of the returned value.

However, if you return condition values rather than signal error conditions, you return less information about the error condition to the calling program. It is recommended that you return condition values when the explanation of the error condition is simple and self-contained. For example, LIB$GET_VM returns a condition value. This is because the possible status conditions are self-contained and simple (for example, insufficient virtual memory).

According to the VAX Procedure Calling and Condition Handling Standard, the status returned must be a VAX condition value. (For further information, see the *Introduction to VAX/VMS System Routines*.)

2.7.3 When to Signal or Return Condition Values

To some degree, whether you decide to signal an error condition or return a condition value depends on the language you are using for your procedure. In some high-level languages it is very difficult to write a condition handler to be invoked in the event that an error condition is signaled. (For further information on condition handling in your language, consult the appropriate language reference manual.)

Regardless of which language you are using, there are general guidelines on when to return a condition value and when to signal an error condition.

You should signal an error condition when:

- Your procedure returns a value in R0 and cannot return a condition value.
- Your procedure must execute quickly and checking the return status of a condition value would be inefficient.
- Your procedure will be executed repetitively and therefore checking the condition value returned would adversely affect your procedure's performance.
- The amount of information you wish to return about the error condition could not be contained in a condition value.
- A useful error message requires information that cannot be determined until run time. For example, the FDL$PARSE procedure must tell you which line of the FDL file was the cause of an error. The line number of the line containing the error cannot be determined until run time. Therefore, the signal mechanism is preferred because the line number information could not be returned in a condition value.
- You wish to execute a specific condition handler in the event that an error condition is signaled.

You should return a condition value when:

- You wish to keep the error-handling mechanism simple.
- The speed of the error checking mechanism is not of great concern.
- The total possible errors that may be returned is a small number and sufficient information about those errors can be contained in the condition value returned.
- The functions provided by the procedure are so general that the procedure will be used in various levels and environments.

3 Coding

This chapter describes how to code modular procedures. It is broken down into the following sections:

- Coding Guidelines
- Initializing Modular Procedures
- Using VAX/VMS System Services
- Invoking Optional User-Action Routines
- Writing AST-Reentrant Code

3.1 Coding Guidelines

The coding guidelines discussed in this section are drawn from the VAX/VMS Modular Programming Standard (Appendix A). These coding guidelines are of two types: *required* and *recommended*. You must follow the sections marked *required* to ensure that your application is modular. DIGITAL highly recommends that you adhere to the guidelines presented in the sections marked *recommended*. Following these additional rules will help you produce consistent, uniform applications.

3.1.1 Writing Position-Independent Code (Required)

A module is position independent when it can execute correctly anywhere in virtual memory. When programming in a VAX high level language, it is difficult not to write position-independent code because of the way the languages are designed. However, when programming in VAX MACRO and BLISS, certain combinations of addressing modes and the address of the operand will result in code that is not position independent.

For more information on writing position-independent code in VAX MACRO, see Section 3.2.2 of the *VAX/VMS Linker Reference Manual*.

3.1.2 Adhering to the Naming Conventions

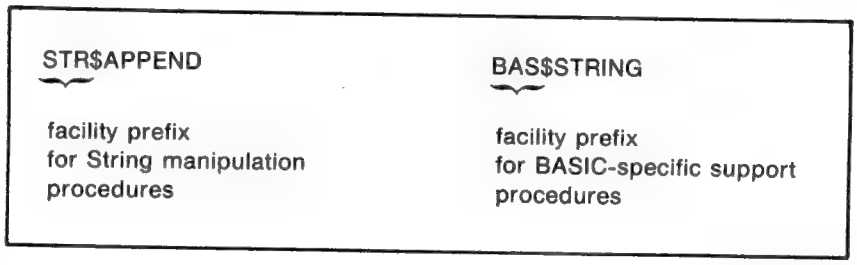
The following guidelines apply to the naming of facilities, procedures, files, modules, and program sections (PSECTs). It is required that you follow these conventions when choosing names for modules, PSECTs and status codes. It is highly recommended that you also adhere to the other naming conventions.

3.1.2.1 Facility Naming Conventions (Recommended)

To make it easy to locate a set of related procedures, we recommend that you group your procedures into facilities. Providing related procedures with a common facility prefix is a convenient method for organizing procedures. The facility prefix is the first part of any procedure name.

As shown in Figure 3–1, the first three characters of a procedure name are used as to indicate the facility of a Run-Time Library procedure.

Figure 3–1 Examples of Facility Prefixes as Used in Procedure Names



ZK-3084-84

Facility names represent library facilities. A procedure is characterized as belonging to a particular facility dependent upon the types of operations it performs. Facilities may differ in the conventions they use for handling errors and receiving arguments, as well as differing in the primary function of the facility. Table 3–1 lists some common DIGITAL facility prefixes.

Table 3–1 Common Library Facilities — Prefixes and Content

Prefix	Content
ADA	Ada Run-Time Library procedures
APL	APL Run-Time Library procedures
BAS	BASIC Run-Time Library procedures
B32	BLISS-32 Run-Time Library procedures
CDU	Command Definition Utility
CLI	Command Language Interpreter
COB	COBOL Run-Time Library procedures
COR	CORAL Run-Time Library procedures
C74	COBOL-74 Run-Time Library procedures
DBG	Debugger
DBL	DIBOL Run-Time Library procedures
ERF	Error Log Formatter
FDV	FMS Forms Driver Library procedures
FOR	FORTRAN Run-Time Library procedures
LBR	Librarian Utility procedures
LIB	RTL General Purpose procedures
MTH	RTL Mathematics procedures
OTS	RTL Language-Independent procedures
PAS	PASCAL Run-Time Library procedures
PLI	PL/I Run-Time Library procedures
RMS	Record Management Services
RPG	RPG II Run-Time Library procedures
SMG	RTL Screen Management procedures
SOR	Sort Utility procedures
STR	RTL String Manipulation procedures
VAX	VAX Architecture Emulation

You can create your own facilities by defining a unique facility name and facility number. The name for your facility should be a unique name between 1 and 27 characters in length. We recommend that you choose facility names between 2 and 4 characters. DIGITAL-supplied facility names all contain a dollar sign ($) after the prefix. User-supplied facility names should

use an underscore (_) rather than a dollar sign ($) to avoid any name conflicts.

The facility number is used in defining VAX condition values for the facility. Bit 27 (STS$V_CUST_DEF) of a condition value indicates whether the value is user or DIGITAL supplied. This bit must be 1 if the facility number is user created. For additional information, see the *VAX/VMS Message Utility Reference Manual*. The Message Utility is used in creating VAX condition values and their associated text.

3.1.2.2 Procedure Naming Conventions (Recommended)

When you create a procedure and make its name global, you allow other procedures in the same image to call that procedure. The common Run-Time Library procedures are examples of procedures with global names. In such an environment, a naming convention is required to prevent any name conflict between global procedures in the same image.

The rules for naming entry points to procedures have the general form:

```
fac$symbol (DIGITAL-supplied)
fac_symbol (user-supplied)

  fac = a two to four character facility name.
  symbol = a symbol from one to 27 characters long.
           (The entire procedure name may not exceed
           31 characters in length.)
```

The facility name and symbol name are separated by a dollar sign ($) if the procedure is DIGITAL supplied, and by an underscore (_) if the procedure is user supplied. This convention should be used to avoid conflict between DIGITAL and user procedure names.

The procedure name usually consists of a verb and an object. Together, the verb and object describe the action of the procedure. For example, the Run-Time Library procedure that is intended to get virtual memory is called LIB$GET_VM.

Some procedures, even though they have global names, are not intended to be called from outside the facility in which they are located. These procedures are available only internally, within a set of procedures, and do not by themselves provide any functionality for the facility. The names for these procedures contain a double dollar sign ($$) if they are DIGITAL supplied,

or a triple underscore (___) if they are user supplied. (Three underscores are necessary to avoid conflict with user-defined condition value symbols which use two underscores.)

The names in Table 3–2 are examples of procedure entry points names.

Table 3–2 Naming Procedure Entry Points

Procedure Name	Description
LIB$GET_VM	DIGITAL-supplied global procedure
LIB_PRINT_REPORT	User-supplied global procedure
OTS$$INTERNAL	DIGITAL-supplied internal procedure
LIB___ADD_TAX	User-supplied internal procedure

3.1.2.3 File Naming Conventions (Recommended)

You should derive your file name from the name(s) of the procedure(s) contained in the module which comprises the file.

If a module contains a single procedure, the file name consists of the procedure name. You may, if you wish, remove dollar signs ($) and underscores (_), but this is not required. File types are the standard default file types for the source language. For example, the file containing the VAX/VMS Run-Time Library procedure MTH$EXP is named MTHEXP.MAR. This name makes it obvious that the file MTHEXP.MAR contains the procedure MTH$EXP and is written in VAX MACRO.

Sometimes, the module which composes the file will contain more than one procedure. For example, the VAX/VMS Run-Time Library procedures LIB$GET_VM and LIB$FREE_VM are contained in the same module, and thus, the same file. In this case, a more general file name is used. This name is composed of the facility prefix (LIB) and the first nouns common to all procedure names in the module (VM). Thus, the name for the file containing procedures LIB$GET_VM and LIB$FREE_VM is LIBVM.B32. (The file type B32 indicates that the module is written in VAX BLISS32.)

3.1.2.4 Module Naming Conventions (Required)

Module names are identical to file names except module names do not have extensions, and the dollar sign ($) or underscore (_) which separates the facility prefix and symbol name is not removed.

For example, the MTH$EXP procedure is contained in module MTH$EXP and the file MTHEXP.MAR. The LIB$GET_VM and LIB$FREE_VM procedures are contained in the module LIB$VM and the file LIBVM.B32.

3.1.2.5 PSECT Naming Conventions (Required)

The code and data sections of a customer library procedure have two separate program sections (PSECTs), named _fac_CODE and _fac_DATA, where *fac* is the facility name. DIGITAL uses _fac$CODE and _fac$DATA as PSECT names.

Position-independent constant data is in the PSECT named _fac_CODE (_fac$CODE for DIGITAL) to shorten the references. For example, _LIB$CODE and _LIB$DATA are the only two PSECT names used by LIB$ procedures.

The collating sequence for leading underscores causes the linker to place all library procedures after the user program in the executable image. This prevents a library procedure from being placed between two user modules and adversely affecting any byte or word displacement addressing that the user program contains.

Not all VAX languages give you control over PSECT names. In VAX BASIC, VAX FORTRAN, and VAX PASCAL, it is not possible to control PSECT names except through use of COMMON. Use of COMMON violates the VAX/VMS Modular Programming Standard.

For additional information about declaring PSECTs, see the appropriate language reference manual.

3.1.2.6 Lock Resource Naming Conventions (Recommended)

When using the VAX/VMS Lock Manager, the resource names of root level locks (locks without a parent) should be derived from the facility name. The naming convention used is:

```
fac$name = DIGITAL-supplied resource name
fac_name = user-supplied resource name
```

Following this convention will prevent unintended resource conflicts.

3.1.2.7 Global Variable Naming Conventions (Recommended)

Global variables should be named using the following format:

```
fac$Gt_variablename = DIGITAL-supplied global variable name
fac_Gt_variablename = user-supplied global variable name
```

The letter "t" indicates the contents and usage of the global variable. The possible values of t are listed in Table 3–3.

Likewise, the format for addressable global arrays is as follows:

```
fac$At_variablename = DIGITAL-supplied global variable name
fac_At_variablename = user-supplied global variable name
```

Again, the letter "t" indicates the contents and usage of the addressable global array. The possible values of t are listed in Table 3–3.

Table 3–3 Code for the Content and Usage of Global Variables

t	Content and Usage of Global Variable
A	Address
B	Byte integer
C	Single character
D	D_floating
E	Reserved to DIGITAL
F	F_floating
G	G_floating
H	H_floating
I	Reserved for integer extensions

Table 3–3 (Cont.) Code for the Content and Usage of Global Variables

t	Content and Usage of Global Variable
J	Reserved to customers for escape to other codes
K	Constant
L	Longword integer
M	Field mask
N	Numeric string (all byte forms)
O	Octaword
P	Packed string
Q	Quadword integer
R	Records (structure)
S	Field size
T	Text (character) string
U	Smallest unit of addressable storage
V	Bit field
W	Word integer
X	Context dependent (generic)
Y	Context dependent (generic)
Z	Unspecified or non-standard

3.1.2.8 Status Code and Condition Value Naming Conventions (Required)

The format of status codes and condition values is as follows:

```
fac$_status = DIGITAL-supplied status code or condition value
fac__status = user-supplied status code or condition value
```

3.1.3 Using Common Source Files (Recommended)

For some applications, it may be necessary to make identical argument declarations in several modules. VAX languages allow you to centralize these declarations in one place by using common source files. Table 3–4 summarizes the common source file declarations for VAX languages.

Table 3–4 How to Declare Common Source Files

Language	Common Source File Declaration
Ada	To share common declarations among Ada programs, you include the declarations in a package (as a separate compilation unit) and provide visibility to the package by using a WITH clause in programs that want to share the common declarations.
BASIC	You can use the BASIC %INCLUDE directive in your program to include the common source file, or a CDD record.
BLISS	Your source program can contain a REQUIRE or LIBRARY list option that specifies a file to be included at the point of the declaration.
COBOL	The COPY statement specifies source text from a COBOL library file, a VAX Librarian file or a VAX Common Data Dictionary record description which is to be included in the source program.
FORTRAN	The INCLUDE statement specifies a file or library module to be included at the point of the statement. You may also use a CDD record.
MACRO	An auxiliary source file or macro library can be specified in the command line, or by using a CDD record.
PASCAL	The %INCLUDE directive and INHERIT attribute specify files to be included at the point of the declarations. You may also use a CDD record.
PL/I	The %INCLUDE preprocessor statement specifies a file to be inserted as source. You may also use a CDD record.
RPG II	An auxiliary source file can be specified in the command line.

3.1.4 Maintaining Code Readability

While you are coding your new application, it is important to remember that at some future time either you or someone else will probably have to update it. To make these inevitable updates easier, concentrate on making your code easy to read and understand.

The following guidelines will help you to produce code that is easily read and understood:

- Use symbols in place of numbers
- Use uppercase and lowercase characters
- Add optional spaces
- Insert block comments

3.1.4.1 Using Symbols in Place of Numbers (Recommended)

Use symbols, not numbers, as much as possible. Because symbols are mnemonic, they will make your programs clearer and provide more information for cross-reference listings. It is always recommended that you define a symbol for a constant that will be used a number of times. If the value for that symbol changes in the future, you will only have to change that value where it is defined, not in every place that it is used in the program. Thus, using symbols instead of numbers makes procedures easier to maintain. Furthermore, if you use a symbol instead of a number, it's easy to see each place that the symbol is referenced in the program. The cross-reference listing for a symbol will list each place that the symbol is reference in the program.

The following figure shows the information that can be obtained about a symbol in a cross-reference listing. Figure 3–2 shows the contents of the LIGHT.LIS file created by the following command:

```
$ PASCAL/LIS/CROSS LIGHT.PAS
```

Note that the cross-reference listing shows where the symbols are declared and where they are referenced.

Figure 3–2 A Sample Cross-Reference Listing Showing the References to the Symbol SPEED_OF_LIGHT

```
MAX_SPEED
01                                  Source Listing

0001                          PROGRAM MAX_SPEED;
0002                    {+}
0003                    {  This program shows the use of a symbol
0004                    {  instead of a number to improve clarity.
0005                    {  The program declares a symbol for the
0006                    {  speed of light, and then uses the symbol
0007                    {  to calculate the speed of the Starship
0008                    {  Lollipop and the Villain Vessel.
0009                    {-}
0010
0011                          CONST
0012                              SPEED_OF_LIGHT = 3 * (10**8);
0013
0014                          VAR
0015                              LOLLIPOP_SPEED  : INTEGER;
0016                              VILLAIN_SPEED   : INTEGER;
0017
0018                          BEGIN
0019                              LOLLIPOP_SPEED := 8 * SPEED_OF_LIGHT;
0020                              VILLAIN_SPEED := 4 * SPEED_OF_LIGHT;
0021                          END.
0022


MAX_SPEED
01                                  Cross Reference Listing

LOLLIPOP_SPEED          VAR   INTEGER { IN PROGRAM MAX_SPEED }
                                  15        19 =

INTEGER                   TYPE
                                  *         15        16

VILLAIN_SPEED             VAR   INTEGER { IN PROGRAM MAX_SPEED }
                                  16        20 =

MAX_SPEED                 PROG  [PSECT($CODE)]
                                  1

SPEED_OF_LIGHT            CONST 300000000 { IN PROGRAM MAX_SPEED }
                                  12        19        20
```

3.1.4.2 Using Uppercase and Lowercase Characters (Recommended)

You should use uppercase characters for all source code except comments. You should use both uppercase and lowercase characters for all comments. Comments that are complete sentences should start with a capital letter and end with a period.

All source code must be coded in uppercase characters if you want your applications to be operating system independent. Although VAX/VMS and its language implementations are insensitive to case, other operating systems and language implementations are not insensitive to case. (Note that VAX C is an exception; VAX C is case sensitive.) Applications coded in mixed or lowercase characters may not be easily transportable to other systems.

The PASCAL program shown in Example 3–1 illustrates the proper use of uppercase and lowercase characters.

Example 3–1 PASCAL Program Showing Use of Uppercase and Lowercase Characters in Code

```
{+}
{  Program to call LIB$LP_LINES and determine the
{  number of lines per line printer page.
{-}
PROGRAM LINES(OUTPUT);
{+}
{  Declare the external procedure used by this
{  program.
{-}
FUNCTION LIB$LP_LINES : INTEGER; EXTERN;
{+}
{  Call LIB$LP_LINES and print the result.
{-}
BEGIN
    WRITELN('Each page contains ',LIB$LP_LINES,' lines.');
END.
```

3.1.4.3 Adding Optional Spaces (Recommended)

A single space should follow a comma (,) and precede and follow an equal sign (=). A single space should precede a left parenthesis (() or a left square bracket ([) (except in MACRO), but not a left angle bracket (<). A space should also follow an exclamation mark (!) or semicolon (;) to separate a comment from the source code. The arithmetic operators plus and minus (+ and -) should be surrounded by spaces in expressions.

The BASIC program in Example 3–2 shows the proper use of optional spaces.

Example 3–2 BASIC Program Showing Use of Optional Spaces in Code

```
10      !+
        ! The following BASIC program converts a character
        ! string representing a hexadecimal value to a
        ! longword, then adds one to the result.
        !
        ! Declare the external routine used.
        !-
        EXTERNAL LONG FUNCTION OTS$CVT_TZ_L
        !+
        ! Perform the conversion.
        !-
        HEXVAL$ = "80012BFA"
        RET_STAT% = OTS$CVT_TZ_L (HEXVAL$, HEX% )
        !+
        ! Add one to the result.
        !-
        HEX% = HEX% + 1
        END
```

3.1.4.4 Inserting Block Comments (Recommended)

You can comment on blocks of statements by writing one or more lines preceding the block. The first comment line contains a single plus sign (+); the last comment line contains a single minus sign (–). Block comments do not need to be set off by additional blank lines; the two flag lines starting with a plus sign (+) and minus sign (–) serve that purpose. However, you may still wish to set off block comments with blank lines to improve readability. Comment delimiters are followed by one space, except when followed by the first plus sign (+) and the last minus sign (–), as shown in Example 3–3.

Example 3–3 FORTRAN Program Showing Use of Block Comments in Code

```
!+
!       This program demonstrates a call to the
!       Run-Time Library procedure STR$PREFIX.
!
!       Initialize the strings to be used.
!-
        A$ = "ABC"
        B$ = "DEF"
!+
!       Call STR$PREFIX
!-
        ISTAT = STR$PREFIX (A$, B$)
        END
```

3.1.5 Using VAX/VMS System Services

Not all VAX/VMS system services are modular, according to the definitions in this manual. Procedures that call nonmodular system services are nonmodular themselves. If your procedure uses a nonmodular system service, you should list the system service in the "Side Effects" section of the procedure description. (For information about the procedure description, see Section 2.5.2.) For further information about particular system services and modularity, see the *VAX/VMS System Services Reference Manual.*

3.1.6 Invoking Optional User Action Routines

An optional user action routine is a useful way to let the calling program gain control at a critical point in your procedure's algorithm. The most common user action routines are success routines and error routines. Control would be passed from your procedure to the optional error routine if the specified error was encountered within your procedure. To transfer control, the calling program must pass the user action routine as an argument to the called procedure. To make it easy for the calling program to pass information to its action routine your procedure should supply an optional **user-arg** argument that the calling program can pass to its action routine. Your procedure merely copies the argument list entry of the user

argument, if present, to the argument list it passes to the action routine. This achieves the same effect as up-level addressing.

There are two VAX data types that may be used to pass a user action routine as an argument: (1) the *procedure entry mask*, and (2) the *bound procedure value*. To provide an interface to a user action routine for your procedure, you must first decide whether to use the *procedure entry mask* or the *bound procedure value* data type. The contents of the argument list entry for your action routine will differ, depending on whether you are specifying a *procedure entry mask* or a *bound procedure value*.

3.1.6.1 Procedure Entry Mask

The *procedure entry mask* (DSC$K_DTYPE_ZEM) is the simplest to use. It is used by FORTRAN and most other languages and is expected by the Run-Time Library procedures, Record Management Services, and system services.

For a *procedure entry mask* passed by reference, the argument list entry contains the address of the *procedure entry mask* to be called. To provide a user action routine using the *procedure entry mask*, your procedure should have the following calling sequence:

```
CALL myproc [action-routine [,user-arg]]
```

In this example, **action-routine** is a function call of the procedure entry mask type that is passed by reference, and **user-arg** is unspecified.

3.1.6.2 Bound Procedure Value

The *bound procedure value* (DSC$K_DTYPE_BPV) is used by PASCAL and other languages where context of the procedure must be known. The procedure might do up-level addressing of a variable defined in a syntactically outer block and hence, allocated in another frame. (If you use a *procedure entry mask*, this context is specified in the **user-arg** argument.)

For a *bound procedure value* passed by reference, the argument list entry contains the address of two longwords. The first longword contains the address of the procedure and the second contains the environment pointer to be loaded into R1 before the procedure is called. This environment pointer allows you to specify the context of your action routine so that you may do up-level addressing. To provide a user action routine using the

bound procedure value passed by reference, the calling sequence is as follows:

```
CALL myproc  [action-routine [,user-arg]]
```

In this example, **action-routine** is a function call of the bound procedure value type that is passed by reference, and **user-arg** is unspecified.

If you wish to use the *bound procedure value* data type to pass access a user routine specified as a *procedure entry mask*, then you must pass the first longword by value and omit the second longword. Then, the user action routine would have this calling sequence:

```
status = action-routine (...[,user-arg])
```

In this example, status is a longword condition value that is passed by value, and **user-arg** is unspecified. Your procedure copies the 32-bit argument list entry passed by the calling program to the argument list provided to the action routine. Thus, the calling program and its action routine can communicate using any data type, access type, passing mechanism, or VMS usage.

3.2 Initializing Modular Procedures

Some modular procedures must initialize themselves before they can execute correctly. Following are examples of initialization:

- Storing in static storage a value that can only be determined at run time
- Declaring an exit handler using the $DCLEXH system service
- Allocating a process-wide resource once
- Opening a file the first time the procedure is called

You must perform initialization carefully to maintain modularity.

Initialization must not affect the calling program. Therefore, avoid initializing by providing an entry point that must be called before any other entry point is called. Providing an entry point that must be called first forces the calling program to

provide an initialization entry point to its caller, and so forth. Also, you would have to rewrite your calling programs if you needed to substitute a procedure with an initialization call for one without an initialization call.

If your procedure uses LIB$INITIALIZE, you must preserve a modular environment that does not conflict with the environment set by any other procedure using LIB$INITIALIZE. (See the *VAX/VMS Run-Time Library Routines Reference Manual* for more information.)

Following are several ways to initialize a procedure:

- Initialize at compile or link time.
- Use the mechanism provided by LIB$INITIALIZE to perform initialization once for each image activation.
- Set a first-time flag at run time.
- Initialize storage each time it is allocated at run time.
- Initialize storage each time a procedure is called at run time.

The use of each method is explained in the following sections. Figure 3–3 summarizes these methods.

3.2.1 Initializing Storage

For a procedure to produce predictable results, all statically and dynamically allocated areas must be initialized to known values before they are read. Initialization of dynamically allocated stack and heap data involves writing the data after each allocation and before reading it.

If your procedure has static storage, it is usually initialized to zero. In some languages, for example BLISS, you do not need to explicitly initialize static store. These languages will automatically initialize static storage to zero. To see if the language you are using initializes static storage implicitly, refer to your reference manual for that language.

There are three ways to explicitly initialize storage. You can use an initialization statement, test and set a first-time flag at run time, or use LIB$INITIALIZE. The method of testing and setting a first-time flag is explained in Section 3.3.4.2.

Figure 3–3 Methods of Initializing

Initialization Needed	Method				
	Initialize at Compile/Link Time	Call LIB$INITIALIZE Before Main Program (At Run Time)	Set a First Time Flag (At Run Time)	Initialize Each Time It Is Allocated (At Run Time)	Initialize Each Time Procedure Is Called (At Run Time)
Of Static Storage:	•	•	•		
Of Stack Storage:					•
Of Heap Storage:				•	
To Allocate Resources:		•	•		
To Set Up $EXIT Handler:		•	•		
To Open a Process-Permanent File:		•	•		
To Set Up a Handler Before the Main Program:		•			

ZK-3085-84

Table 3–5 provides examples in the major VAX languages of how to initialize a longword, DAT, in static storage using an initialization statement.

Table 3–5 How to Initialize Static Storage

Language	Statement	Initialized Value	Time of Initialization
Ada	x : INTEGER := 1	1	Elaboration time
BASIC	BASIC does not permit static storage within a module, only common static storage.		
BLISS	OWN DAT;	0	Compile time
	OWN DAT INITIAL(0);	0	Compile time
	OWN DAT INITIAL(100);	100	Compile time
COBOL	01 NUM PIC	0	Compile time
	01 NUM PIC 9 VALUE 0.	0	Compile time
	01 NUM PIC 9(3) VALUE 100.	100	Compile time
FORTRAN	INTEGER*4 DAT	0	Compile time
	INTEGER*4 DAT/0/	0	Compile time
	DATA DAT /0/	0	Compile time
	DATA DAT /100/	100	Compile time
MACRO	DAT: .BLKL 1	0	Compile time
	DAT: .LONG 0	0	Compile time
	DAT: .LONG 100	100	Compile time
PASCAL VAR			
	DAT : [STATIC] INTEGER;	0	Compile time
	DAT : [STATIC] INTEGER := 0;	0	Compile time
	DAT : INTEGER : = 100;	100	Compile time
PL/I	STATIC INIT(2)	2	Compile time
	EXTERNAL INIT(3)	3	Compile time
	GLOBALDEF INIT(4)	4	Compile time
	GLOBALREF INIT(5)	5	Compile time
RPG	RPG II has static storage at the module level only. Numeric variables are initialized to zero and alphanumeric variables are initialized to spaces at compile time.		

ZK-3083-84

3.2.2 Testing and Setting a First-Time Flag

To do first-time initialization, your procedure may test and then set to one a statically allocated first-time flag each time it is called. This flag is initialized to zero at compile or link time. Setting and testing the flag with the VAX instruction Branch on Bit Set and Set (BBSS) insures that initialization is executed exactly once. (You may also use BBSSI or the Run-Time Library procedure LIB$BBSSI.)

However, if your implementation language does not have access to VAX instructions, and the procedure is to be AST-reentrant, it must follow these steps:

1. Test the first-time flag.
2. If the first-time flag is set, initialization is complete.
3. If the first-time flag is not set, disable ASTs. Remember the previous state of AST enable, and retest the flag.
4. If the first-time flag is now set, initialization was performed by an AST that occurred between the first test and the AST disable; enable ASTs if remembered state of ASTs was enable. Initialization is now complete.
5. If the first-time flag is not set, perform the initialization.
6. Set the flag.
7. Enable ASTs if remembered state of ASTs was enable—initialization is complete.

For additional information, see Section 3.3.

Note: ASTs should only be enabled (step 4 or step 7) if they were enabled before step 3. The $SETAST system service, used to disable ASTs, indicates whether ASTs were enabled when the procedure was called.

Example 3–4 illustrates the use of a first-time flag in a PASCAL program to allocate a resource.

Example 3–4 PASCAL Program Which Uses a First-Time Flag

```
{+}
{ Program to demonstrate the use of a first-time flag when allocating
{ a resource.  This technique is AST-reentrant, but is NOT multi-thread
{ reentrant.
{-}
PROGRAM ALLOCATE;

CONST
    VM_SIZE = 512;

VAR
    INITIALIZED : BOOLEAN := FALSE;
    VM_ADDRESS :  INTEGER := 0;
    AST_STATUS :  INTEGER := 0;
    VM_STATUS :   INTEGER := 0;
    DISABLE :     INTEGER := 0;

FUNCTION LIB$GET_VM (SIZE : INTEGER; VAR ADDR : INTEGER) : INTEGER; EXTERNAL;
FUNCTION SYS$SETAST (VAR STATUS : INTEGER) : INTEGER; EXTERNAL;
```

(Continued on next page)

Example 3–4 (Cont.) PASCAL Program Which Uses a First-Time Flag

```
BEGIN
    {+}
    { Check the first-time flag.  If set, initialization has been
    { performed already.
    {-}
    IF NOT (INITIALIZED)
    THEN
        BEGIN
        {+}
        { Disable AST's, and remember the previous state.
        {-}
        AST_STATUS := SYS$SETAST (DISABLE);
        {+}
        { Now, recheck the flag.  If it is now set, initialization was
        { performed by another invocation of this procedure between when
        { the flag was first tested and now.  Otherwise, initialization
        { is performed here.
        {-}
        IF NOT (INITIALIZED)
        THEN
            BEGIN
            {+}
            { Perform the initialization.
            {-}
            VM_STATUS := LIB$GET_VM (VM_SIZE, VM_ADDRESS);
            {+}
            { Set the first-time flag, indicating initialization complete.
            {-}
            INITIALIZED := TRUE;
            END;
        {+}
        { Restore AST's to the previous state.
        {-}
        AST_STATUS := SYS$SETAST (AST_STATUS);
        END;
END.
```

3.2.3 Using LIB$INITIALIZE

One of the ways that you may initialize a value at run time is by using the Run-Time Library procedure LIB$INITIALIZE.

If your procedure establishes a condition handler by calling LIB$INITIALIZE before a main program, the action of this handler may conflict with other condition handlers established by other procedures before the main program.

However, there are certain circumstances under which it may be necessary to initialize a value at run time. An example of a value that you may need to initialize at run time is a seed for a random number generator.

There are six steps needed to use LIB$INITIALIZE to initialize a value at run time:

1. Write the main program.
2. Write an initialization procedure.
3. Write a MACRO or BLISS program to add the address of that initialization procedure to PSECT LIB$INITIALIZE.
4. Compile the initialization procedure, main program, and MACRO program.
5. Link the initialization procedure, main program, and MACRO program.
6. Run the main program.

Assuming that you have completed the main program, the first thing that you need to do is write an initialization procedure. If, for example, you were going to use LIB$INITIALIZE to initialize a value for a random number generator, you might write an initialization procedure to set the seed equal to the current time. This would generate a different seed for each initialization because the time is constantly changing. One possible initialization procedure is shown in Example 3–5.

Example 3–5 BASIC Initialization Procedure for LIB$INITIALIZE

```
100  !+
     ! Initialization routine.  A common piece of data, called SEED,
     ! is initialized based on the number of CPU seconds used by
     ! this process so far.
     !-
     SUB MY_INIT_ROUTINE(ONE,TWO,THREE,FOUR,FIVE,SIX)
     COMMON (MY_DATA) LONG SEED
     PRINT "Now in initialization routine."
     CURRENT_TIME = TIME(1)
     SEED = CURRENT_TIME
     END SUB
```

Once you have defined the initialization procedure, you must write the MACRO program to add the address of that initialization procedure to PSECT LIB$INITIALIZE. The format for this MACRO program is very simple, as seen in Example 3–6.

Example 3–6 Program to Add Address to PSECT LIB$INITIALIZE

```
;+
; Make references to external routines used.
;-
        .EXTRN  LIB$INITIALIZE
        .EXTRN  MY_INIT_ROUTINE
;+
; Make a contribution to the PSECT LIB$INITIALIZE.
;-
        .PSECT    LIB$INITIALIZE USR,GBL,NOEXE,NOWRT,LONG
        .ADDRESS  MY_INIT_ROUTINE
        .END
```

To modify this MACRO program for use in your own procedures, simply substitute the name of your initialization procedure for MY_INIT_ROUTINE.

Once you have written the initialization procedure and the MACRO program to add the dispatch address to PSECT LIB$INITIALIZE, you simply link and run your program. We will use the main program shown in Example 3–7.

Example 3–7 BASIC Main Program

```
10      !+
        ! Mainline.  The value of SEED is printed.
        ! The linker initializes this value to zero, but because LIB$INITIALIZE
        ! is used, an initialization routine is run before control is transferred
        ! here, and the value of SEED is changed to a somewhat random value.
        !-
        COMMON (MY_DATA) LONG SEED
        PRINT "Now in mainline.  The seed is initialized to: ";SEED
32767   END
```

To run LIB$INITIALIZE and thus initialize the value of SEED at run-time, enter the following commands:

```
$ BASIC MAIN
$ BASIC INIT
$ MACRO LIBRARY
$ LINK MAIN,INIT,LIBRARY
$ RUN MAIN
```

One example of the output generated by these steps is as follows:

```
Now in initialization routine.
Now in mainline. The seed is initialized to: 4099
```

3.3 Writing AST-Reentrant Code

This section describes coding techniques for modular procedures that use the VAX/VMS asynchronous system trap (AST) interrupt mechanism, or permit calling programs to use it.

All modular procedures should be AST-reentrant so they can be called from any program. If your procedure is not AST-reentrant or calls any procedure that is not, your program documentation should specify this to warn others against using your procedure.

3.3.1 What is an AST?

An asynchronous system trap (AST) is a VAX/VMS mechanism for providing a software interrupt when an external event occurs. One example of this type of interrupt occurs when the user types CTRL/C. When the external event occurs, the VAX/VMS operating system interrupts the execution of the current process and calls a procedure that you supply. This procedure is what we refer to as the AST handler.

Some VAX/VMS system services let an external event interrupt a process. It is because the interrupt occurs out of sequence with respect to process execution, that the interrupt mechanism is called an "asynchronous" system trap. The AST interrupt transfers control to the AST handler that services the event. This AST handler can call other procedures, including library procedures.

The AST handler which you provide and any procedures it calls are said to be executing at AST level. While at AST level, a process cannot be interrupted a second time at the same access mode. The process runs to completion at the AST level before the non-AST level procedure resumes.

A process is either executing at AST level or at non-AST level, and thus consists of two "threads of execution," one thread at each level. Keep in mind that these levels are threads of the same process, and not separate processes.

When your AST handler finishes servicing the event, it returns control to its caller. The interrupted procedures continues execution from the point of interruption.

For example, you could call the Set Timer system service ($SETIMR) to specify the address of an AST-level procedure to be executed after a specified amount of time has elapsed. At the specified time, the system will generate an AST interrupt by stopping the procedure that is currently executing and calling the specified AST handler.

For information on the implementation of AST interrupts by system services, see the *VAX/VMS System Services Reference Manual.*

3.3.2 AST-Reentrancy Versus Full-Reentrancy

A procedure is AST-reentrant if it meets the following conditions:

- It can be interrupted at any point, permitting itself or any related procedure to be called (reentered).
- It will execute correctly when it continues from the point of interruption.

Do not confuse the term *AST-reentrant* with the term *fully-reentrant*. Full reentrancy refers to a more restrictive set of conditions.

In an AST-reentrant environment, the AST thread is expected to complete regardless of whether it encounters a locked resource. When the AST thread encounters a locked resource in an AST-reentrant environment, it expects to be given a "new" resource, or it is expected to return an error message. It is never expected to wait for the resource that the non-AST level has locked.

In a fully-reentrant environment, all threads are treated equally when they encounter a locked resource; they will wait for the resource to be freed. In a fully-reentrant environment, AST threads are not given any special treatment. The VAX Ada environment is an example of a fully-reentrant environment. In such a situation, there can be more than two threads of concurrent execution, and each thread can alternately progress toward an end.

Note: It is highly desirable that future code satisfy the more stringent requirement of being fully-reentrant. Full reentrancy is important for procedures which will be called from multi-thread environments, such as Ada tasks. For further information, refer to the VAX Ada documentation.

3.3.3 Guidelines for Writing AST-Reentrant Modular Procedures

To use AST interrupts, you must write an AST handler to take control at AST level. An AST handler can be written in any language. Because the particulars of writing an AST handler differ from one language to the next, see the reference manual for the language you are working in for more details.

In general, an AST handler must follow these guidelines:

- It must be separate from the procedure that is currently executing.
- It must not modify data or instructions used by the interrupted procedure or its callers.
- The AST handler must be callable with a CALLG or CALLS instruction, and must return with a RET instruction.
- If it modifies any registers other than R0 and R1, it must set bits in the entry mask to save the contents of the registers.
- If it calls any other procedures, they must all be AST-reentrant.
- The AST handler cannot stall or use "busy wait" to avoid being called before the non-AST level is out of a critical section of code. Once the AST handler has begun executing it cannot be interrupted by anything at a non-AST level. In fact, the only thing that can interrupt the AST handler is another procedure running at AST level in a different access mode.

  If you attempt to use a "busy wait" and expect to change the condition from the non-AST level, the AST level will circle the "busy wait" in an infinite loop. The process will continue to loop because the non-AST level will not continue executing until the AST thread has finished and thus will never be able to change the value in the "busy wait" condition.
- You cannot use the lock manager to protect a resource that is being accessed at non-AST level from being accessed at AST level. The lock manager is designed to lock resources between separate processes, not different threads (AST and non-AST) of the same process.

- Avoid using static storage. A procedure that does not use static storage, calls only AST-reentrant procedures, and does no up-level addressing is automatically AST-reentrant.

3.3.4 How to Eliminate Race Conditions During Concurrent Access

There are two problems that you may encounter in using AST interrupts. They are called race conditions and deadlocks. A race condition occurs when your AST-handler attempts to use a nonshareable resource that is already in use by the non-AST thread of execution.

If you allow the AST handler to wait for the resource (for example, by waiting for an event flag to be set by the non-AST level code of the same access mode) you have caused a form of deadlock. The deadlock occurs because the non-AST level code cannot execute to free the resource until the AST-level code has finished executing. The AST level code cannot continue either because the non-AST level code has effectively locked the resource.

A race condition occurs when you attempt to access or modify the same data in static storage by both the AST and non-AST level of a process. For example, if an AST begins executing while the non-AST level is modifying data in static storage, that data may be left in a non-stable state while the AST handler executes. To prevent a race condition you should allow only one thread at a time to modify data.

If a procedure does not modify any static storage, then it is both AST-reentrant and fully-reentrant. There are a number of ways for your procedure to eliminate conflict when accessing and modifying data in static storage:

- Perform all accessing or modification in a single uninterruptable instruction. (This technique also makes the procedure fully-reentrant.)
- Detect concurrency of access to data using "test and set" instructions at entry to and exit from data storage. The procedure may then report an error, or retry the operation (when appropriate) if concurrency is detected.

- Place a branch on bit clear and then clear (BBCC or BBCCI) instruction immediately after your procedure has completed access to static storage.

The BBSS instruction detects that a concurrency conflict is about to take place before static storage has been accessed. If the storage is being accessed by multiple processors, you must use BBSSI and BBCCI.

There are two alternate techniques for resolving concurrency conflicts detected by the BBSS and BBCC instructions.

- Use separate, statically allocated areas for storage at the AST and non-AST levels. When the BBSS instruction detects concurrency at the beginning, use the second allocated area. Note that this technique does not work if an exception condition occurs between execution of the BBSS instruction and the BBCC instruction or if your procedure has not established a condition handler. This is because a condition handler established by the calling program might also simultaneously call your procedure.
- Reexecute your procedure if concurrency is detected. When the BBCC instruction detects this concurrency, branch back to the beginning of your procedure and try again.

Example 3–9 illustrates the latter technique. This MACRO procedure, LIB_GET_INUM, allocates and deallocates identifying numbers:

Example 3–9 MACRO Program Showing Use of Test and Set Instructions

```
.TITLE  LIB_GET_INUM  -- Allocate and deallocate id. nos. 1 - 10
TAB:      .WORD   0                   ; Bitmap for flags
          .ENTRY  LIB_GET_INUM, ^M<>
10$:      FFC     #1, #10,TAB, R0     ; Find first free id, no.
          BEQ     20$                 ; Branch if none free
          BBSS    R0, TAB, 10$        ; Indicate id.  no. in use
          MOVL    R0, @4(AP)          ; Return id. no. found
          MOVL    #1, R0              ; Indicate success
          RET
20$:      CLRL    @4(AP)              ; Return 0
          CLRL    R0                  ; Indicate failure
          RET
          .END
```

3.3.4.3 Keeping a Call-in-Progress Count

If the database is to be kept separate between calls, you can keep track of when your procedure is called by using a call-in-progress count. Before database access, the count is incremented and used as an index for an address table of the separate databases. You should check for a count that exceeds the table length. After the database has been accessed, the count is decremented.

This technique has an advantage over the BBxx technique because it can handle more than two levels of reentrance. However, it is less reliable, because an exception can cause the count never to be decremented, leading to an eventual procedure malfunction. You can avoid this by establishing a condition handler in your procedure.

3.3.4.4 Disabling AST Interrupts

A procedure is also considered AST-reentrant if AST interrupts are disabled while critical sections of code execute. However, this method of maintaining AST reentrancy is not recommended.

Sometimes the only way to avoid race conditions is to disable AST interrupts during the access to static storage and restore the state of the AST enable once the critical section of code has finished executing. However, this technique could adversely affect performance of real-time programs using AST interrupts. The $SETAST system service which is used to enable and disable AST interrupts is time consuming. Therefore, you should avoid disabling AST interrupts whenever you can by using the techniques described in Sections 3.3.4.1 to 3.3.4.3.

Try to minimize the number of instructions during which the AST interrupts are disabled. Before disabling AST interrupts, establish a condition handler to restore the AST level in case an exception or stack unwind occurs.

Example 3–10 demonstrates how $SETAST can be used to disable ASTs and then restore the previous state of the enable.

Example 3–10 A FORTRAN Program Disabling and Restoring ASTs

```
!+
! This program demonstrates using the System
! Service SYS$SETAST to disable and then
! re-enable AST-interrupts.
!-
        INCLUDE '($SSDEF)'
        INTEGER*4 SYS$SETAST
!+
! Turn off ASTs and remember the previous setting.
!-
        ISTAT = SYS$SETAST (%VAL(0))
!+
! The statements in the program during whose
! execution you want ASTs disabled.
!
! If ASTs were previously enabled,
! re-enable them.
!-
        IF (ISTAT .EQ. SS$_WASSET) CALL SYS$SETAST( %VAL(1))
        END
```

3.3.5 Performing Input/Output at AST Level

If your procedure performs I/O using VAX RMS system services, there are several coding techniques you must follow for your procedure to be AST-reentrant:

- When opening process-permanent files such as SYS$INPUT, SYS$OUTPUT, SYS$COMMAND, or SYS$ERROR, check for the VAX RMS error status RMS$_ACT (active) after each $CREATE or $OPEN service. This error indicates that a record operation has already started for the process-permanent file. The error does not occur for files which are not process permanent, and the $OPEN service follows the constraints of shared access to the file that may have been imposed by a previous $OPEN service. If the error occurs, perform a $WAIT using the same file access block (FAB). When control returns to your procedure, try the $CREATE or $OPEN service again. Repeat this sequence until it succeeds.

- When performing record I/O to any type of file, check for the RMS error status RMS$_RSA (record stream active) after each $GET and $PUT service. This error indicates that a record operation has already been started for the file. If the error occurs, perform a $WAIT using the same record access block (RAB). When control returns to your procedure, try the $GET or $PUT service again. Repeat this procedure until it succeeds.
- Avoid storing data in a record access block (RAB) that VAX RMS could still be accessing. You can avoid this situation by doing either of the following:
  - Allocate the RAB on the stack so the AST and non-AST level have separate RABs.
  - Allocate RAB in static storage along with a busy bit. The busy bit is tested and set using a BBSS instruction before the RAB is accessed. If the RAB is already busy, your procedure executes a $WAIT using that RAB.

  For synchronous I/O (I/O that is always completed before returning control to your procedure), you can allocate the RAB in either of these ways. However, the first method is more reliable, since it doesn't use static storage and therefore will not become corrupted if an exception is signaled.

  For asynchronous I/O (when control is returned to your procedure before I/O is completed), you must use the second technique.

3.3.6 Condition Handling at AST Level

You should not allow an exception to propagate out of an AST handler since the exception may be caught by any procedure that is active at the time of the AST. Condition handlers for other active procedures may react as if the exception was caused by a procedure that they had called.

Another reason for not allowing exceptions to propagate out of an AST handler is that for run-time environments which use multiple threads in a process, it cannot be determined which stack (of the threads of execution) is used to deliver the AST. (The AST is delivered on the stack of whichever thread is active at the time of the AST interrupt.)

It is best to catch all exceptions in the AST handler and not allow them to propagate.

4 Testing

A successful test system is one that uncovers errors. To facilitate successful testing you should begin planning for the testing phase during the design phase. Preliminary testing should start while you are coding the procedure. In this respect, some parts of the testing stage are concurrent with the coding stage of the software life cycle.

The two primary things that you should be testing for are:

1 To make sure that the procedure you developed fulfills your requirements/specifications.

  You must carefully test each aspect of functionality to ensure that your procedure does everything that it was intended to do and nothing that it was not intended to do. The methods that you use to test this aspect of your procedure will depend upon the function(s) of your procedure.

2 To make sure that the procedure is modular and executes without error.

This chapter focuses on testing for modularity.

Modularity is especially important to procedures that will be included in a library facility. If your procedure is in any way non-modular, it may adversely affect the results and performance of other procedures which call it. It is essential that your procedure completely adheres to the required sections of the VAX/VMS Modular Programming Standard contained in Appendix A.

To ensure modularity within your procedures, you should perform at least three types of tests:

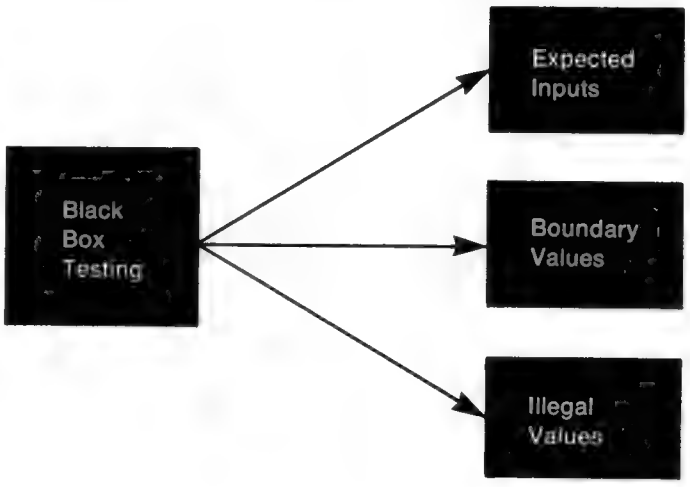
1 Unit testing

2 Language-independence testing

Using the example above, what will happen if your procedure receives as input a value that is less than 1 or greater than 999? Will the user receive a useful error message? Will the procedure simply stop, or will it attempt to use values outside of its limitations and simply return an incorrect answer? It is essential that you run the procedure using illegal input values to determine the answers to these questions.

Figure 4–1 summarizes the methods of black box testing.

Figure 4–1 The Methods of Black Box Testing



4.1.2 White Box Testing

When performing white box testing, unlike black box testing, you must understand fully the internal workings of the procedure. Those internal workings, those specific lines of code, are in fact what you are testing.

There are three steps involved in white box testing.

1 Test each statement.

For this step, you need to provide sets of test values that ensure that every statement in the procedure is executed at least once. This includes all statements—even those executed only when optional arguments, user-supplied arguments, subroutines, user-action routines, or specific error codes are present.

2 Test each decision.

At this step, your goal is to provide test cases that ensure that each branch of a decision is executed at least once. In the case of a standard boolean decision, this will generally require providing two values, however, this number may be much greater in the case of compound or nested decisions.

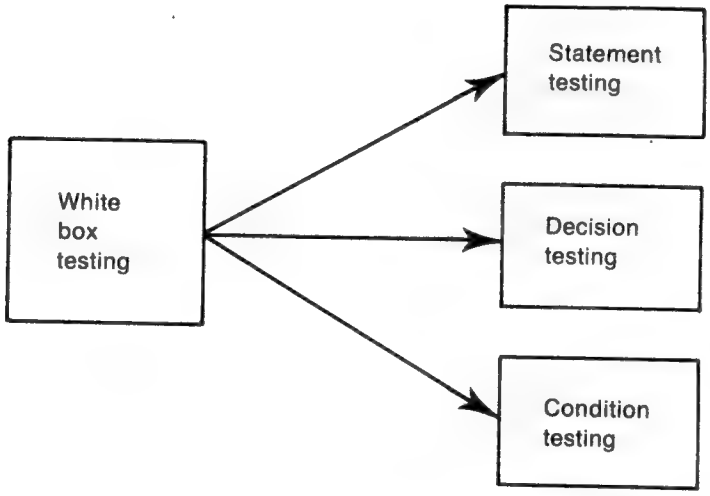
3 Test each condition.

Condition testing requires writing test cases that ensure that each condition in a decision takes all possible outcomes at least once and that each point of entry to the program or subroutine is invoked at least once. Multiple test values must be supplied in cases of compound and nested loops. In testing the entry points, remember to invoke any optional routines (either internal or external), as well as error handlers. If your procedure contains a JSB entry point, that entry point should also be tested.

Note that each type of white box testing finds a specific type of error. For example, statement testing will not find an error on a negative value for a condition if the statement is given a positive input the only time it is executed. Therefore, you must perform all three types of white box testing.

Figure 4–2 summarizes the methods of white box testing.

Figure 4–2 The Methods of White Box Testing



4.2 Language-Independence Testing

For your procedures to be as useful as possible, they must be able to be called by programs in any language. Providing for language independence is essential to produce a useful procedure.

Testing for language independence is a very specific type of unit testing. It ensures that your program executes correctly regardless of what language it is called from.

To test your procedures for language independence, write several driver programs in languages you have chosen randomly. The driver program need only contain a call to the procedure being tested.

If you do find that your procedures are not language independent, make sure that they conform to the following rules:

- All atomic data must be passed by reference and all strings must be passed by descriptor.

  Adherence to this single guideline is the most important factor in achieving language independence.

- Statements which assume a particular language environment are NOT allowed.

  For example, the statement ON ERROR GO BACK in a BASIC procedure assumes that the calling program is also written in BASIC.

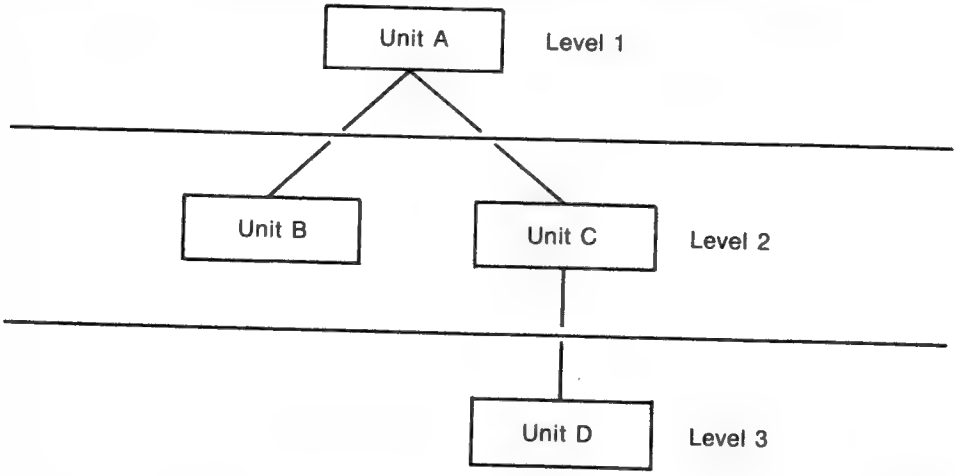
4.3 Integration Testing

Integration testing is the next logical step following unit testing. Unit testing is designed to test each separate component. That component might be a module, a subprogram, a subroutine, an internal procedure (fac___name), or a particularly intrinsic piece of code depending on your procedure. Once you have determined that each unit works separately, you need to determine that the units also work together to form the complete procedure.

4.3.1 The "All at Once" Approach to Integration Testing

Integration testing can be completed by one of two methods. The first method is the "all at once" approach. In this method of integration testing, you simply finish all of the units, link them together and test the completed structure all at once. Use of this method is strongly discouraged, because it makes it very difficult to find the location of errors. For example, look at the organization of the units in the sample procedure shown in Figure 4–3. Let us assume that we tested this procedure using the "all at once" approach and found an error. The procedure simply did not work. We would have no way of knowing whether the error was in unit A, unit B, unit C, or unit D.

Figure 4–3 A Sample Procedure for Integration Testing



4.3.2 The Incremental Approach to Integration Testing

The recommended approach to integration testing is called *incremental* testing. Incremental testing involves testing the procedure by starting with one unit and building on to it one unit at a time. Each unit should always be subjected to thorough unit testing before it is included in the integration tests.

Incremental integration testing is especially useful for finding the following types of errors:

- Problems with the calling interface between units (for example, inconsistent ordering of arguments between the calling and called unit)
- Incorrect assumptions about what values are returned and which units they are returned to
- Unexpected transfer of control between units

Using the sample procedure in Figure 4–3, you would complete the test of unit A on level 1 before proceeding to level 2 where you would test units A and B in combination. At each level you would correct any errors before proceeding to the next level. When you have completed this last step, testing Units A, B, C, and D, you know that the entire procedure works correctly.

Because you started at the top of the sample procedure and added units incrementally from lower levels, you were using the *top-down* approach to integration testing. You could just as easily have started at Level 3 and used the *bottom-up* approach.

As you can see from the example, there are several distinct advantages to incremental integration testing:

- It is not necessary to wait until the procedure is complete to begin integration testing.
- Debugging is simplified by incremental testing because the modules and interfaces can be tested as the system grows.
- Programming errors in the interfaces and incorrect assumptions between units are discovered at an early stage.
- Because previously tested units are retested as new units are added, the probability of discovering less obvious errors is increased substantially.

4.4 Testing for Reentrancy

It is important to test your procedures for reentrancy before placing them into a library facility. Because ASTs can occur at any time, procedures which are not AST-reentrant may exhibit unexpected behavior. In particular, an AST which occurs during storage modification in a procedure which is not AST-reentrant may corrupt the contents of the procedure's storage. (For further information about AST reentrancy, see Section 3.3.)

Full reentrancy is important to multi-thread tasking environments such as the environment used by Ada.

To avoid problems with reentrancy, carefully read and follow the coding guidelines described in Section 3.3.

4.4.1 Checking for AST Reentrancy

There are two methods of checking a procedure for AST reentrancy. You can use the Symbolic Debugger or perform a manual *desk check*.

4.4.1.1 Checking for AST Reentrancy using the Debugger

There are five steps necessary to use the Debugger to check for AST reentrancy. Those steps are as follows:

1. Create an activation of the procedure.
2. Set watchpoints on all storage used by the procedure.
3. Create a second activation of the procedure using the CALL command. Allow this second activation to run to completion. (The second activation represents the AST-level thread.)

   Check to be sure that the AST-level thread of execution does not modify the storage accessed by the non-AST level thread of execution. If the AST-level thread of execution does modify any of that storage, check to ensure that it does not cause any unwanted side effects for the non-AST level thread of execution.
4. Step one instruction in the first activation.
5. Repeat steps 3 and 4 until the end of the procedure for the first activation.

For further information about the Debugger, refer to the *VAX/VMS Utilities Reference Volume*.

4.4.1.2 Checking for AST Reentrancy by Desk Checking

Desk checking is the term for tracing through a procedure's execution manually. Performing a desk check for AST reentrancy consists of the following four steps:

1. Create an activation of the procedure being tested and its data using the method you standardly use for manually tracing through a procedure.

   This activation represents the non-AST level of your procedure's execution.

2. Create a second activation of the procedure using the process you used above. This second activation represents the AST-level thread of your procedure's activation.

   Trace through the AST-level thread's execution to completion, one statement at a time.

   Remember to update the contents of all storage locations and variables for each instruction of the procedure.

   Check to be sure that the AST-level thread of execution does not modify the storage accessed by the non-AST level thread of execution. If the AST-level thread of execution does modify any of that storage, check to ensure that it does not cause any unwanted side effects for the non-AST level thread of execution.

3. Step through a single statement of the non-AST level thread of execution, remembering to update the contents of all storage locations.

4. Repeat steps 2 and 3 until you have stepped through every statement in the non-AST level thread of execution. (Note that every statement of the AST-level thread is stepped through in each pass through step 2.)

As you can see, what you are actually doing in the process is testing between the execution of every two statements in the procedure. The most rigorous method of applying this type of desk checking for AST reentrancy is to step through the procedure at the assembly language level and test between each assembly language instruction.

4.4.2 Checking for Full Reentrancy

Full reentrancy differs from AST reentrancy in the number of threads of execution. An AST-reentrant environment can support only two threads of execution, the AST-level thread and the non-AST level thread. Full reentrancy is important in environments that can support many threads of execution, such as Ada.

A procedure is fully-reentrant if any number of threads of execution can execute to completion without affecting any of the other threads of execution.

Generally, a procedure that is AST-reentrant is also fully reentrant. For further information on full reentrancy and environments supporting multiple threads of execution, refer to the documentation for VAX Ada.

4.5 Performance Analysis

All timer and resource allocation procedures should make statistics available for performance evaluation and debugging. You should code timer and resource allocation procedures with the following two entry points:

```
LIB_SHOW_name
LIB_STAT_name
```

4.5.1 SHOW Entry Point

A SHOW entry point provides formatted strings containing the information you need. The calling sequence for a SHOW entry point is as follows:

```
LIB_SHOW_name [code [,action-routine [,user-arg]]]
```

code

An optional code (of the form LIB_K_code) designating the statistic you need. Define a separate code for each statistic available; the codes should be the same for the SHOW and STAT entry points. The values associated with the codes start at one for each procedure. The functional specification in the procedure's documentation should list the codes used. If the code is omitted, or zero, the procedure provides all statistics.

action-routine
The address of an action routine. This is an optional argument. If omitted, statistics are written to SYS$OUTPUT.

user-arg
An optional user argument to be passed to the action routine. If omitted, a shortened list is passed to the action routine. The **user-arg** argument, if present, is copied to the argument list passed to the action routine. That is, the 32-bit argument list entry passed by the calling program is copied to the argument list entry passed to the action routine. Thus, the access type, data type, argument form, and passing mechanism can be arbitrary, as agreed between the calling program and the action routine.

The optional action routine should have the form:

```
ACTION-ROUTINE (string [,user-arg])
```

See Section 3.1.6 for an example of the code to invoke a user action routine.

4.5.2 STAT Entry Point

A STAT procedure returns the information you want as binary results. The calling sequence is as follows:

```
LIB_STAT_name (code ,value)
```

code
A code designating the statistic you want. A separate code is defined for each statistic available; the codes are the same for the SHOW and STAT entry points. Codes start at one.

value
Is the value of the returned statistic.

4.6 Monitoring Procedures in the Run-Time Library

There are several procedures available in the Run-Time Library for time and resource monitoring. These Run-Time Library procedures and their functions are as follows:

- LIB$SHOW_VM

  LIB$SHOW_VM is a resource monitoring procedure that returns the statistics accumulated from calls to LIB$GET_VM and LIB$FREE_VM.

  The information returned by default is the following three statistics:

  1 Number of successful calls to LIB$GET_VM

  2 Number of successful calls to LIB$FREE_VM

  3 Number of bytes allocated by LIB$GET_VM but not yet deallocated by LIB$FREE_VM

  LIB$SHOW_VM returns these statistics in the formatted form, nnnn.

- LIB$STAT_VM

  LIB$STAT_VM is a resource monitoring procedure that returns to its caller one of the three statistics available from calls to LIB$GET_VM and LIB$FREE_VM. These are the same statistics that are returned by LIB$SHOW_VM. Unlike LIB$SHOW_VM which returns the statistics in formatted form to SYS$OUTPUT, LIB$STAT_VM returns the specified statistic in a signed longword integer.

- LIB$SHOW_TIMER

  LIB$SHOW_TIMER is a time monitoring procedure that returns the times and counts accumulated since the last call to LIB$INIT_TIMER and displays them on SYS$OUTPUT. A user-supplied action routine may alter this default behavior.

The information provided by default is the following five statistics:

1. Elapsed real time
2. Elapsed CPU time
3. Count of buffered I/O operations
4. Count of direct I/O operations
5. Count of page faults

- LIB$STAT_TIMER

  LIB$STAT_TIMER is a time monitoring procedure that returns the same information as LIB$SHOW_TIMER. The difference is that LIB$STAT_TIMER returns the information as an unsigned longword or quadword whereas LIB$SHOW_TIMER returns the information in the format hhhh:mm:ss:cc for times and the format nnnn for counts. In addition, LIB$STAT_TIMER returns only one of the five available statistics per call.

For further information about these time and resource monitoring procedures, see the *VAX/VMS Run-Time Library Routines Reference Manual*.

5 Integration

5.1 Grouping Procedures

Modular procedure libraries consist of compiled and assembled object code intended to be associated with a calling program at link time. References to procedures in these libraries are resolved when the linker searches the user libraries specified in the LINK command or the default system libraries. The program can then call library procedures at run time.

DIGITAL supplies several procedure libraries, such as the VAX Common Run-Time Procedure Library (also called the Run-Time Library) that support components of the VAX/VMS operating system. You can explicitly access the procedures in the Run-Time Library to perform frequently used operations. To do this, simply include calls to Run-Time Library procedures in your program. The linker automatically searches the default libraries to resolve references to Run-Time Library procedures. (For information on what procedures are available in the Run-Time Library, see Section 1.3.1.)

You can create your own procedure libraries and shareable images by following the guidelines in Sections 5.1.2 and 5.2.1. Section 5.2.2 shows you how to use transfer vectors to make your shareable images easier to maintain. Before you begin grouping modular procedures, make sure they conform to the rules listed in Appendix A.

There are three ways to group modular procedures:

1 Combine object modules into an object module library.

2 Link object modules together into a shareable image.

3 Combine shareable images into a shareable image library.

The following sections show how to create and install these three types of procedure libraries, and how to access them when you link and run a program.

5.1.1 Creating Facility Prefixes

The facility prefix is the group identifier for a set of related procedures contained in a library facility. The facility prefix appears in the procedure name of every procedure in that library facility. An example of a library facility is the Screen Management Facility in the Run-Time Library. The names of all procedures appearing in the Screen Management Facility begin SMG, for example, SMG$ERASE_CHARS.

To create your own facility prefix, follow these steps:

1 Choose a facility prefix. This prefix can be from 1 to 27 characters in length. However, it is recommended that you choose facility prefixes between 2 and 4 characters.

2 If your facility will be generating messages, you must specify a unique facility number in the message source file. This number can range from 0 to 4095. Any number that is within this range and is not being used by someone else on your system is acceptable. This facility number will be used by the message utility in generating the condition value for the message.

   Bit 27 (STS$V_CUST_DEF) of a condition value indicates whether that value is user or DIGITAL supplied. This bit must be 1 if the facility number is user created. For further information, see the *VAX/VMS Message Utility Reference Manual.*

3 Use the facility prefix when naming all procedures within the new facility. Remember to follow the naming conventions described in Section 3.1.2.

5.1.2 Creating Object Module Libraries

In addition to using the system default object module libraries, you can also create your own object module libraries. An object module library that you create can contain object files produced by any VAX language compiler.

There are three steps required to create an object module library:

1 Write or collect the procedures that you want to group together in the object module library. Make sure that the names chosen for the modules, procedures, and facility prefixes conform to the guidelines given in Section 3.1.2.

2 Compile the source code to create the object files for the procedures to be contained in the library.

3 Create the object module library using the LIBRARY command.

The format of the library command is as follows:

```
$ LIBRARY /CREATE library-name.OLB file-spec1.OBJ -
_$ [,file-spec2.OBJ [,...]]
```

The parameter **library-name** is the name that you have given the library. The default file extension for **library-name** is OLB.

The **file-spec** parameters are the object files for the procedures you want the object module library to contain. The default file extension for **file-spec** is OBJ.

To clarify this process, let us create a sample object module library.

The first step is to choose the procedures that will be contained in the sample library. The sample library we will create is called GRAPHICS.OLB. This library will contain modular procedures for creating mathematical representations of circles, cylinders, squares, and other geometric shapes.

The files to be used are GRASPHERE.BAS and GRACUBE.FOR. GRACUBE.FOR contains a single procedure, GRA_CUBE, to generate cube shapes. (Note that GRA is our facility prefix, and the underscore in the procedure name indicates that this is a user-defined procedure.) GRASPHERE.BAS contains several

procedures that are grouped together because they share similar code. The procedures contained in GRASPHERE create spheres (GRA_SPHERE), oblate spheriods (GRA_OBL_SPH), and spherical sections (GRA_SPH_SEC).

Now that we have chosen the procedures and named them correctly, we advance to step 2 and create the object files necessary to build the object module library. The commands used to create the object files are as follows:

```
$ BASIC GRASPHERE.BAS
$ FORTRAN GRACUBE.FOR
```

This produces the object files GRASHPERE.OBJ and GRACUBE.OBJ.

The final step is to create the object module library itself from those object files. We create the object module library by issuing the following command:

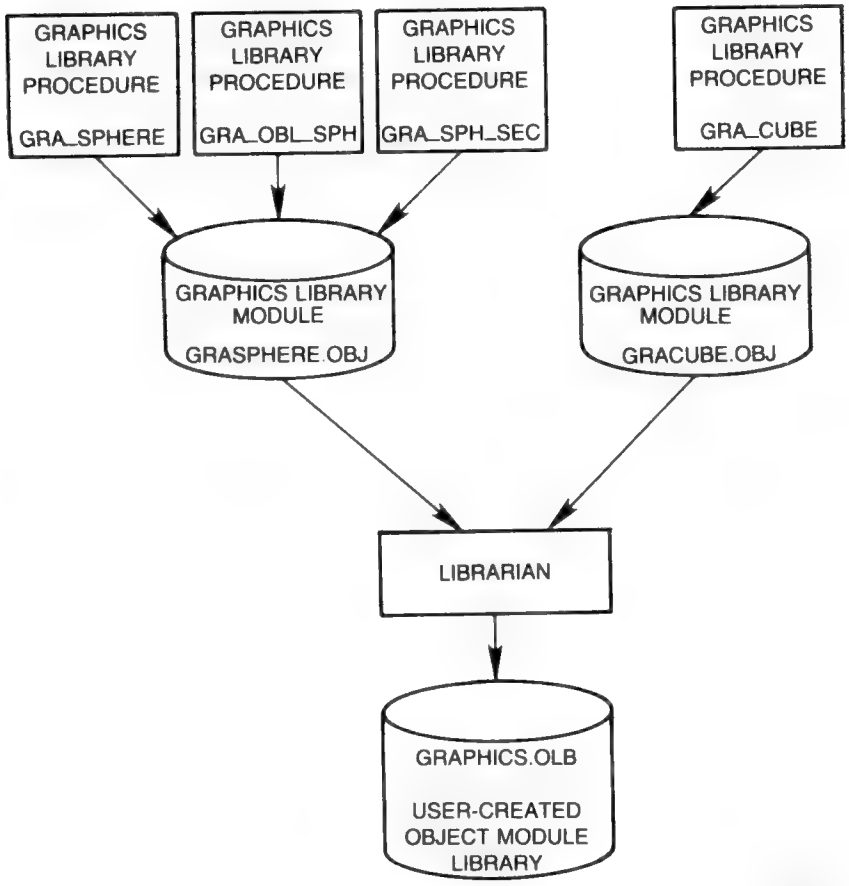
```
$ LIBRARY /CREATE GRAPHICS GRASPHERE,GRACUBE
```

Note that we did not specify the file extensions because we used only the default values.

Once the LIBRARY command has been issued, the object module library GRAPHICS.OLB is ready to be linked with an application program. Linking to libraries of modular procedures is described in Section 5.5.

Figure 5–1 shows the overall development of the user-created library of graphics procedures, GRAPHICS.OLB.

Figure 5–1 Development of a User-Created Object Module Library



5.2 Shareable Library Images

If you have a collection of procedures which you expect will be used by a number of users, you may want to group these procedures into a shareable library image. A shareable library image, usually referred to as a shareable image, is similar to an object library except that it has been pre-linked so that all

references between procedures in the library have already been resolved.

A shareable library image gives you the following advantages:

- Conserves memory space

  Several processes can "share" a single copy of a shareable image rather than each process retrieving its own copy from the disk.

- Conserves disk storage space

  Programs linked to a shareable library image share a single disk copy of the library code rather than each program including the code in its own executable image.

- Shortens link time

  Since the internal references in the library have already been resolved, there is less work for the linker.

- Allows for updates without relinking

  You can supply a new version of a shareable library image which will automatically be used by all programs linked to it without the need for the users to relink their programs.

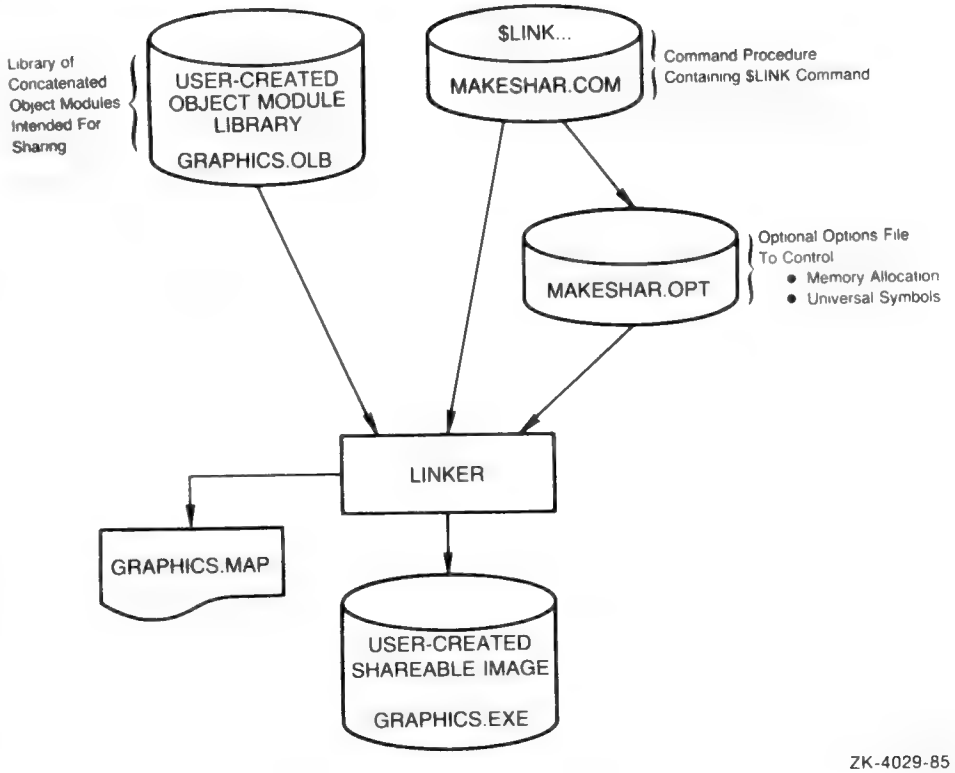
The greatest benefit of using a shareable library image is the ability to conserve physical memory by sharing a single memory copy of the library code among all users. This feature is enabled by having a privileged user, usually the system manager, use the VAX/VMS Install Utility to install the shareable library image with the /SHAREABLE attribute. This benefit is not limited to shareable library images, however. Any executable image can be installed as shareable. See the *VAX/VMS Install Utility Reference Manual* for more information.

5.2.1 Creating Shareable Library Images

It is quite simple to take a collection of procedures and link them together as a shareable library image. In addition to your procedures, you need to provide a transfer vector and a linker options file. The following sections will show you how to create both of these items in a step-by-step fashion. If you follow the "recipe", you will be rewarded with a shareable image library that is easy to use and maintain.

Figure 5–2 provides an overview of the process of creating a shareable image.

Figure 5–2 Creating a Shareable Image



5.2.2 Creating the Transfer Vector

When a program links to a shareable library image, the linker stores two items of information in the program about each reference to a routine in that image.

1 The name of the shareable image

2 The location of the routine entry point relative to the beginning of the shareable image

Since a relative location is used, the routine cannot change its location without making all images using it invalid. Using a transfer vector allows you to modify code and rearrange routines without invalidating previously linked programs.

A transfer vector is a module containing a list of "forwarding addresses" for routines in the shareable image, which is placed at the beginning of the image. When a transfer vector is used, programs linking to the shareable image reference the entry points in the transfer vector rather than the actual routines. The entry in the transfer vector then transfers control to the actual routine. Because each entry in the transfer vector is the same size, it is easy to keep the relative location of an entry the same across updates.

A transfer vector must be created using the VAX MACRO assembler language, but it is not necessary to know VAX MACRO to create a transfer vector. A template for a transfer vector is provided in Figure 5–3. To use this template in creating a transfer vector, follow these steps:

1 Create a file with file type MAR containing the template. A suggested name for the file would be the module name. Therefore, if you were to use the example module name GRA_VECTOR as shown in the transfer vector template, the file name would be GRA_VECTOR.MAR.

2 Follow the instructions in the template and edit the file to produce the correct vector for your facility.

3 Compile the vector module with the DCL command:

```
$ MACRO GRA_VECTOR
```

Figure 5–3 Transfer Vector Template

```
;+
; VAX MACRO template for a transfer vector.
;
; In VAX MACRO, comments begin with a semicolon.
; Blanks and tabs may appear interchangeably wherever
; a blank is shown.
;-

;+
; The following two lines define the name of the module,
; which in this example is GRA_VECTOR (the vector for
; the GRA_ facility), and the identification (version) of
; this module, which is 1-001. Replace GRA_VECTOR by
; whatever module name you choose.
;-

        .TITLE  GRA_VECTOR
        .IDENT  /1-001/

;+
; The following lines define a MACRO called ROUTINE which,
; when invoked, generates one transfer vector entry. Enter
; these lines exactly as shown; no customization is necessary.
;-

        .MACRO  ROUTINE NAME
        .EXTRN          NAME
        .ALIGN  QUAD
        .TRANSFER       NAME
        .MASK           NAME
        JMP             NAME+2
        .ENDM
```

(Continued on next page)

Figure 5–3 (Cont.) Transfer Vector Template

```
;+
; The next two lines define the PSECT
; in which the transfer vector is to reside. The only item
; you might choose to change is the name of the PSECT,
; $$$$VECTOR. If you change the name, make sure that it
; is alphabetically before ANY other PSECT in your shareable
; library image. The use of dollar signs here, normally restricted
; to DIGITAL-supplied software, is necessary because the dollar
; sign sorts alphabetically before all other symbol characters.
; This name cannot interfere with DIGITAL-supplied software,
; so there is no risk to using the dollar signs. The remainder
; of the two lines define PSECT attributes which you should
; not change.
;-
        .PSECT $$$$VECTOR  PIC, USR, CON, REL, LCL, SHR, -
                           EXE, RD, NOWRT, QUAD
;+
; Following this point are the transfer vector entries.
; Each entry is of the form:
;
;       ROUTINE  routine-name
;
; where routine-name is the name of the routine for which you
; wish to make an entry. The order in which you list the routines
; is not important, but it is important that you not change the
; order once you have created the shareable library image.
; If you want to add routines, add them to the end of the list.
; For compatibility, you should never delete a routine.
;
; As examples, entry declarations for the routines GRA_SPHERE,
; GRA_OBL_SPH and GRA_SPH_SEC are shown. Replace these by your
; own entries.
;-
        ROUTINE GRA_SPHERE
        ROUTINE GRA_OBL_SPH
        ROUTINE GRA_SPH_SEC
;+
; The last line denotes the end of the module.
;-
        .END
```

5.2.3 Creating the Linker Options File

A linker options file contains instructions to the linker about how it should build the executable or shareable image. The use of a linker options file is rarely necessary for executable images, because the linker's defaults are usually adequate. However, shareable images need additional information which can only be supplied by an options file.

A linker options file can specify many different image attributes, but only a few of these are usually needed for shareable library images. Figure 5–4 shows a linker options file template, and briefly describes each option that is used. To customize this linker options file for your own needs, follow the instructions in the options file comments. A suggestion for the name of the file would be the facility prefix followed by "_OPTIONS", with a file type of OPT. Thus for the example GRA facility, the options file name would be GRA_OPTIONS.OPT.

Figure 5–4 provides a template for a linker options file.

Figure 5–4 Template for a Linker Options File

```
!+
! In a linker options file, comments begin with an exclamation
! point. Blanks and tabs may appear interchangeably wherever
! a blank is shown.
!-

!+
! The first lines defines the name of your shareable image.
! In this example the name is GRA_SHR (the shareable image for
! the GRA facility). Replace GRA:SHR with the name of your
! shareable image.
!
! If you don't specify the image name, the Linker will use the
! name of the first input file.
!-

NAME = GRA_SHR

!+
! The next line specifies the identfication string. The
! identification string is the version number of the shareable
! image. Replace "V1.0" with the identification string of
! your shareable image.
!
! (You can display this identification string later by using
! the ANALYZE/IMAGE command.)
!-

IDENTIFICATION = "V1.0"

!+
! The following line specifies the global section match criteria
! and identification.
!
! In this example, the global section match criteria is "LEQUAL"
! (less than or equal). You should also use "LEQUAL" in your
! linker options file.
!
! The major and minor global section identification values are
! "1,1". The first "1" is the major identification value. Do
! NOT change this number. The second "1" is the minor
! identification value. Increase the minor identification value
! by one every time you add entries to the transfer vector.
! Do NOT change this value if you are only modifying code without
! adding new entries. (See the VAX/VMS Linker Reference Manual
! for more information.)
!-

GSMATCH = LEQUAL,1,1
```

(Continued on next page)

Figure 5–4 (Cont.) Template for a Linker Options File

```
!+
! If you have blocks of data defined with COMMON (FORTRAN, BASIC,
! PASCAL), MAP (BASIC), GLOBALDEF (PL/I), PSECT_OBJECT (Ada), or
! similar declarations in other languages, you must change the
! SHR attribute of the PSECTs (Program Sections) defined by these
! language features. If you do not change the SHR attribute to
! NOSHR, you will receive an error message.
!
! The PSECT_ATTR statement is needed for PSECTs that have both
! the WRT (writeable) and SHR (shareable) attributes. In this
! example, we assume that there is a COMMON named GRA_COMMON.
! Replace GRA_COMMON with the appropriate PSECT name, or delete
! this statement if there are no appropriate PSECTs.
!-
PSECT_ATTR = GRA_COMMON,NOSHR

!+
! To make sure that the transfer vector comes first in
! the image, all read-only PSECTs must have PIC and EXE
! attributes. Examine the listings from your compilations
! to determine the attributes of the various PSECTs. In
! this example, the PSECT $PDATA, produced by FORTRAN, needs
! to be made PIC and EXE.
!-
PSECT_ATTR = $PDATA,PIC,EXE

!+
! The next two lines group the code and data in your
! shareable library image image section clusters. An
! image section cluster is a group of related PSECTs.
! Put all of your read-only code and data (including
! the transfer vector) in CLUSTER1. Put all read-write
! data in the CLUSTER2. COMMON (or similar) PSECTS
! must also be placed in CLUSTER2.
!
! Replace $$$$VECTOR by the PSECT name you
! specified in your transfer vector, if it is different.
! The PSECT names shown here are typical of those generated
! by high-level languages. Check your compilation listings
! to note any other PSECT names used.
!-
COLLECT = CLUSTER1,$$$$VECTOR,$CODE,$PDATA
COLLECT = CLUSTER2,$LOCAL,$BLANK,GRA_COMMON
```

(Continued on next page)

Figure 5–4 (Cont.) Template for a Linker Options File

```
!+
! The following lines list the modules that are to be
! included in the shareable image library. This
! example assumes that all of the necessary object
! modules are in an object module library called
! GRA_OBJLIB.OLB.
!
! You must explicitly specify the transfer vector object
! file name. The transfer vector will automatically
! reference all defined entry points and any references
! needed.
!
! If you need to reference more modules or libraries, add new
! lines containing the names of the appropriate files.
!-
GRA_VECTOR.OBJ
GRA_OBJLIB.OLB/LIBRARY
!+
! End of linker options file.
!-
```

For more information about linker options files and the linking process, see the *VAX/VMS Linker Reference Manual.*

5.2.4 Creating the Shareable Library Image

Once you have created the transfer vector and the linker options file, you are ready to link the shareable library image. Using the example of a shareable library image for the GRA facility, which we will call GRA_SHR, the command to link the image is as follows:

```
$ LINK/SHAREABLE=GRA_SHR.EXE/MAP=GRA_SHR.MAP/FULL GRA_OPTIONS/OPTIONS
```

Substitute your own image name and linker options file name for the names GRA_SHR and GRA_OPTIONS above. The linker will create the shareable library image as GRA_SHR.EXE, and will also create a link map file, containing useful information about the linked image in a file named GRA_SHR.MAP. If you also want a cross-reference of symbols used in your shareable library image, add the command qualifier /CROSS_REFERENCE to the command line.

The link should complete without any errors or other messages being displayed. There is one warning that may occur if any of the modules in your library are written in either VAX MACRO or VAX BLISS-32. (Programs written in high-level languages will not exhibit this problem.) The text of the message is "basing image due to errors in relocatable references." While the link operation will complete successfully, and the resulting shareable library image can be used, you will lose the desirable attribute of upwards compatibility when you link a new image or when an image referenced by your shareable library image is relinked. The cause of this error is references to external symbols without using general mode addressing. In VAX MACRO, prefix all external references with G^; in VAX BLISS, either declare the external name with the attribute ADDRESSING_MODE (general) or use the SWITCHES statement to force general-mode addressing for externals. See the *VAX MACRO and Instruction Set Reference Manual* and the *BLISS Language Guide* for more information.

Once you have linked your image, examine the linker map to make sure that your transfer vector, in this example PSECT $$$$VECTOR, has been allocated at a base address of zero. If it has not, you will probably need to change the attributes of some PSECTs to PIC and EXE, as shown in the template linker options file. See Section 5.3.4.1, Generation of Image Sections, in the *VAX/VMS Linker Reference Manual* for additional information.

5.2.5 Combining Shareable Images into a Shareable Image Library

Given a collection of shareable images, you can create a shareable image library using the library command. The format of the library command used to create a shareable image library is as follows:

```
$ LIBRARY /CREATE /SHARE library-name -
_$ shareable-image1 [,shareable-image2 [,...]]
```

The parameter **library-name** is the name that you have given the shareable image library. The **shareable-image** parameters are the shareable images you want the shareable image library to contain.

5.3 Linking to Libraries of Modular Procedures

To use a library of modular procedures, whether in an object module library or a shareable image library, specify the name of the library along with the /LIBRARY file qualifier, as one of the input files on the LINK command. For example:

```
$ LINK MYPROG,OURLIB/LIBRARY
```

In this example, MYPROG.OBJ is the main program of what is to be an executable image and OURLIB.OLB is either an object module library or a shareable image library. However, if OURLIB is a shareable image library, and if the shareable library image does not reside in the system directory SYS$LIBRARY:, then you must define a logical name for the shareable image name to point to the file's actual location. For example:

```
$ DEFINE GRA_SHR OURDISK:[OURLIB]GRA_SHR
```

This logical name must be defined before the program is run.

6 Maintenance

6.1 Upward Compatibility

Upward compatibility is a very important concept in the maintenance stage. If a procedure is upwardly compatible, then changes and updates to the procedure will not affect the execution and use of the previous versions of that procedure.

For example, let us imagine a user-written procedure named LIB_TOTAL_BILL. The calling sequence for this procedure is as follows:

```
CALL LIB_TOTALIB_TOTAL_BILL (sale, tax)
```

Let us assume that the customer who wrote this procedure decided to update the procedure so that it could be used to calculate the total bill for credit card customers. To do this, a third argument, **interest**, must be added. To be upwardly compatible, the adding of the argument **interest** must not conflict with the way the procedure was previously run. The new calling sequence would be as follows:

```
CALL LIB_TOTAL_BILL (sale, tax [,interest])
```

The procedure should be written so that the customer can still call the procedure as it was called before; simply omitting the **interest** argument.

If, in the updated version of this procedure, the customer can still follow the calling sequence of the previous versions, the procedure is said to be upwardly compatible.

6.1.1 Making Your Procedures Upwardly Compatible

To be compatible with all future versions of the shareable image, shareable image procedures must adhere to the following rules (in addition to following the VAX/VMS Modular Programming Standard):

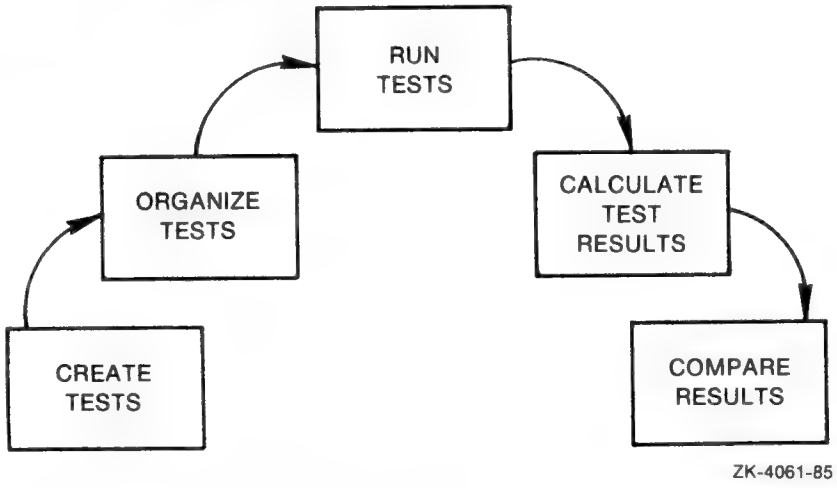
- A procedure's entry point is referenced through a transfer vector. (For further information, see Section 5.2.2.)
- A procedure's code and data is position independent.

6.1.2 Regression Testing

Regression testing is a method of ensuring that new features added to a procedure do not affect the correct execution of previously tested features. In regression testing, you run established software tests and compare test results with the results you expected to get. If the actual results do not agree with what you expected, the software being tested may have errors. If errors do exist, the software being tested is said to have "regressed".

There are six steps in regression testing, as illustrated in Figure 6–1.

Figure 6–1 Regression Testing



1 Create tests

You should create tests by writing command files to test your software.

2 Organize tests

You should organize test files to allow easy access to tests as they are needed.

3 Run tests

To run a single test, submit its command file to the batch queue.

To run multiple tests, create a command file that submits each test to the batch queue.

4 Calculate test results

Calculate the expected test results either by hand or by using previously tested software.

5 Compare results

Compare the actual test results to the results you expected. If there are inconsistencies, repeat your calculation in step 4. If the inconsistency still exists, examine the changes you have made to the software to discover the error.

It is important to write new tests and repeat the regression testing steps every time you add new functionality to the procedure. If you do not repeat the regression tests at each addition of functionality, the procedure may regress while the errors go undetected.

6.1.2.1 Updating the Transfer Vector

There are three steps required to update the transfer vector. These steps are as follows:

1 Add the new entry or entries.

To maintain upward compatibility when updating a procedure library, it is important that you do not disturb the order of existing entries in the transfer vector. You must always add new entries to the end of the transfer vector, and you must not delete or rearrange existing entries. If you carefully follow these guidelines (which are also given in Section 5.2.2) you should have no problems updating the transfer vector, and your shareable image will be upwardly compatible.

2 If you suspect that there is a compatibility problem, you must check the transfer vector.

You can check the transfer vector by comparing the entry addresses found in the listing file generated by compiling the transfer vector with the entry addresses in a listing file from the previous version of the transfer vector. Therefore, you should always keep a copy of previous listing files. In comparing files, make sure that the addresses of the symbols have not changed.

If all symbols in the previous version are contained in the current version and the relative addresses of those symbols have not changed, then the transfer vector has been changed in an upwardly compatible fashion and existing programs using symbols within the transfer vector will continue to be valid.

3 Update the minor global section match identification in the linker options file.

See the linker options file template in Section 5.2.3.

6.2 Adding Arguments to Existing Routines

During the normal course of maintenance, it sometimes becomes necessary to pass new or additional information to an existing procedure rather than create a new procedure. This new information may be passed to the procedure in one of the following two ways:

1 Directly, by adding new arguments to the procedure

2 Using an argument block

6.2.1 Adding New Arguments to the Procedure

There are two rules which you must follow when directly adding new arguments to a procedure:

1 New arguments must be added at the end of the existing argument list.

2 New arguments must be optional.

It is important that new arguments be added at the end of the existing argument list to maintain upward compatibility. If you change the order of the existing arguments by placing the new argument at the beginning or middle of the list, all applications written with the previous version of the procedure will no longer work.

Your procedure should also treat the new argument as an optional argument. If the new argument is required, applications which used the previous version of the procedure will be invalidated.

Because you cannot assume that all previously written applications will be rewritten to include the procedure's new argument, the procedure must test for the argument's presence before attempting to access it. If the procedure does not verify the presence of the new argument and attempts to

access that argument when it is not present, the results will be unpredictable.

The passing mechanism of the new argument must conform to the guidelines established in Section 2.2.1.

6.2.2 Using Argument Blocks

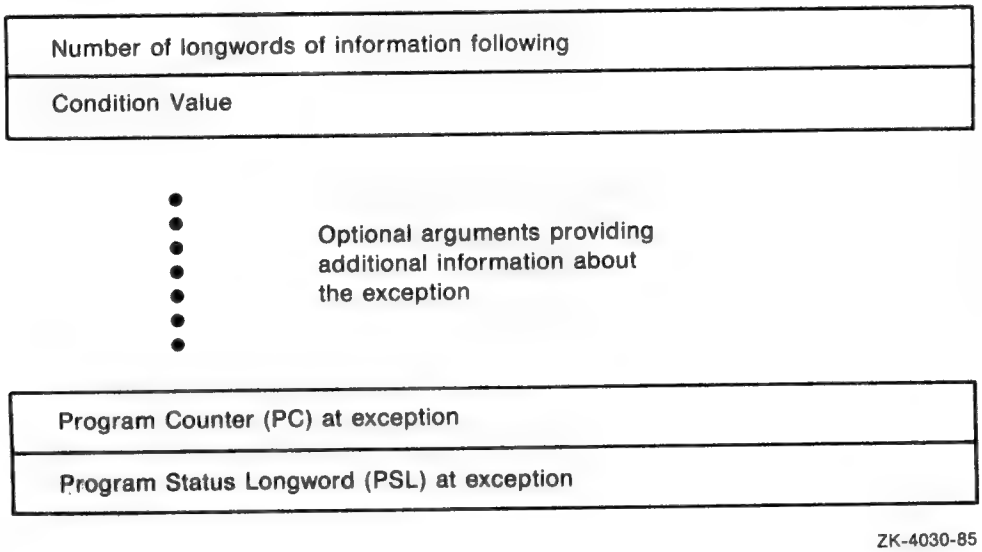
Using an argument block allows you to avoid adding more arguments to your procedure. When you use an argument block, your calling program passes a single argument to the called procedure. This argument is the address of an argument block. The argument block is a block of information containing any information agreed on by the calling and called procedures. This information is required by the called procedure in order to perform its task.

The argument block itself is simply a contiguous piece of virtual memory. The information contained in the argument block may be numeric or scalar data, descriptors, bit vectors, and so on. The format is simply agreed on by the users of the procedure and its writer.

The first longword in the argument block contains the length of the block. The length may be in bytes, longwords, or whatever, but it must be agreed on by both parties concerned and implemented and documented as such.

One example of an argument block is the signal argument vector used in condition handling. A condition handler is called with a signal argument vector and a mechanism argument vector. Each vector is an example of an argument block. The signal argument vector in Figure 6–2 is an example of an argument block.

Figure 6–2 One Type of Argument Block, the Signal Argument Vector



As you can see, the signal argument vector contains the number of longwords of actual information in its first longword. What information actually follows depends on the condition value of the signal.

Note that if you lengthen an argument block to provide new information to a called procedure, your procedure should check the length of the argument block for validity before attempting to access the information. As with adding new arguments directly to a procedure, the calling program may have been written to pass the previous, shorter argument block. If your procedure does not check and attempts to access information past the end of the actual argument block, the results will be unpredictable.

6.3 Updating Libraries

Any time modifications or enhancements are made to modular procedures that are a part of some library, the library containing the procedure(s) must be updated to reflect the new or changed procedures.

6.3.1 Updating Object Libraries

If the updated procedure(s) are in an object library, the library will need to be updated so that subsequent access to that library by LINK or other commands will access the object modules for the new or changed procedures.

To update an object library, use the LIBRARY command with the REPLACE and OBJECT qualifiers, as follows:

```
$ LIBRARY /REPLACE library-name file-spec[,...]
```

In this example "library-name" is the name you have given the library. The default file type for library-name is OLB. The name of an object module is "file-spec". The default file type for file-spec is OBJ.

6.3.2 Updating Shareable Images

If the updated procedure(s) are part of a shareable image, the shareable image will need to be relinked so that it contains the new or changed versions of any updated object modules. If new procedures are added, the transfer vector will need to be updated and recompiled prior to relinking the shareable image. If new modules are added, the linker options file will need to be updated prior to relinking. If new procedures and new modules are added, then the transfer vector and the linker options file will need to be updated. If the transfer vector is changed, the minor identification value of the GS MATCH must be incremented by one. When this has been done, the shareable image may be relinked.

The following sections describe changing the transfer vector and updating the linker options file.

6.3.2.1 Changing the Transfer Vector

It is very important that you change the transfer vector in an upwardly compatible fashion in order to preserve the validity of existing programs that use symbols contained in it.

To avoid invalidating the transfer vector, adhere to the following rules:

- Do not change the order of existing symbols in the transfer vector.
- Do not remove any existing symbols from the transfer vector.
- Add all new symbols to the end of the existing transfer vector.

Recompile the vector module when the changes have been completed, and examine the names and relative addresses assigned to the symbols in the module to ensure that all symbols previously there are still there, and that all relative addresses of those symbols have not changed. If existing symbols are removed or relocated, then existing programs using the shareable image will be invalidated.

6.3.2.2 Updating the Linker Options File

Several items in the linker options file should be changed each time the transfer vector is updated.

- Increment the minor identification portion of the global section match identification string.

  Changing the minor identification value will prevent programs linked against the new version of the shareable image from activating an older version (which may not contain new procedures). Global section match identification values are set with the GSMATCH option.

- If necessary, add any new object modules to be included in the shareable image to the linker options file.
- Change the identification string to a new version number.

6.3.3 Updating Shareable Image Libraries

If the updated procedure(s) are in a shareable image that is part of a shareable image library, the shareable image library will need to be updated so that subsequent access to that library by the LINK command will be to the updated version of the shareable image.

To update a shareable image library, use the LIBRARY command with the REPLACE and SHARE qualifiers, as follows:

```
$ LIBRARY /REPLACE /SHARE library-name file-spec[,...]
```

In this example, "library-name" is the name you have given the library. The default file type for library-name is OLB. The name of a shareable image is "file-spec". The default file type for file-spec is EXE.

A VAX/VMS Modular Programming Standard

This appendix is the VAX/VMS standard for writing modular procedures in any VAX language. This standard is the minimum necessary to interface your software at the callable procedure level with software written by others, and vice versa.

Non-conformance to any elements in this standard must be indicated in your procedure's documentation.

Each element of the standard is described in greater detail in other sections of this manual. References to the appropriate section(s) appear after each element. If following the element is not required to maintain modularity, the word "Optional" appears before the section reference.

A.1 Purpose of this Standard

Most of this standard was derived by asking: "What general agreements are necessary between programmers to permit procedures to execute as expected when combined in arbitrary ways to form a program?"

This means that a procedure not following this standard could cause another modular procedure in the program image to execute incorrectly.

The arbitrary ways of combining procedures are:

- Your procedure calls other procedures.
- Other procedures call your procedure.
- A calling program calls either your procedure or other procedures.

The VAX/VMS Modular Programming Standard was designed to give programmers a common environment in which to write their code. If all programmers follow this standard, then any modular procedure can be added to a procedure library without conflicting with any procedures currently in the library or those that might be added in the future.

A.2 Scope of Applicability

The elements of this standard apply to library procedures and are recommended for other types of software, including utilities and application programs. Each DIGITAL-supplied programming language implemented on the VAX/VMS operating system lets you write your procedures to follow this standard.

This standard applies to procedures that have a public entry point. A public entry point is one that the linker will be able to locate by searching the default system libraries. This standard does not apply to calls to routines internal to a module that do not have public entry points as long as the entire set of procedures follows the standard.

A.3 Coding Rules

The following rules pertain to all procedures. These rules are grouped in the following categories:

- The Calling Interface
- Initialization
- Reporting Exception Conditions
- AST Reentrancy
- Resource Allocation
- The Format and Content of Coded Modules
- Shareable Images
- Upward Compatibility

A.3.1 The Calling Interface

- Calls to procedures must follow the VAX Procedure Calling and Condition Handling Standard. Some elements of this standard restrict procedures to a subset of the VAX Procedure Calling and Condition Handling Standard to increase the ability for procedures to call each other. (See *Introduction to VAX/VMS System Routines.*)
- A procedure makes no assumptions about its environment other than those of this standard. In particular, to operate as specified, a procedure neither makes assumptions about nor places requirements on the calling program.
- A procedure should not call other procedures or system services if the resulting combination violates this standard from the calling program's viewpoint. A procedure can call other procedures or system services that do not follow optional elements of this standard. However, if the resulting combination (as seen from the calling program) does not follow the optional elements, the calling procedure must indicate such non-conformance in its documentation. (See Section 3.1.5.)
- A modular procedure must provide an interface to its callers that allows the callers to follow all required elements of this standard.
- Each module should contain only a single public entry point. (Optional.)
- When a procedure uses a JSB entry point, it should also provide an equivalent call entry point to maintain language-independence. This is because, although JSB calling sequences may execute faster than procedure calls, an explicit JSB linkage to an external routine may not be provided in some high level languages. (Optional. See Section 2.2.6.)
- The order of required arguments should be the same as that of the VAX hardware instructions, namely, read, modify, and write. Optional arguments follow in the same order. However, (according to the VAX Procedure Calling and Condition Handling Standard) if a function value cannot be represented in 64 bits or is of type string, the first argument specifies where to store the function value, and all other arguments are shifted one position to the right. (See Section 2.2.4.)

- A procedure's caller should indicate omitted trailing optional arguments either by passing argument list entries that contain zero or by passing a shortened argument list. However, system services require trailing arguments and do not adhere to this guideline. (Optional. See Section 2.2.5.)
- String arguments should always be passed by descriptor. (See Section 4.2.)
- Procedures must not accept data from or return data to their calling programs by using implicit overlaid PSECTs, or implicit global data areas. All arguments accepted from or returned to the calling program must use the argument list and function value registers (R0 and R0/R1). (See 2.2.2.)
- A procedure cannot assume that the implicit outputs of procedures that it calls will remain unchanged if subsequently used as implicit inputs to those procedures or companion procedures. (See Section 2.2.2.)
- All user code must be position independent. The data need not be position independent. However, for improved performance, data should be initialized to zero at compile or link time to avoid either position-independent constants or position-dependent addresses. (See Section 3.1.1.)
- Position-independent references (in a module) to another PSECT must use longword relative addressing so the VAX Linker can correctly allocate the data PSECT anywhere with respect to the code PSECT no matter how many code modules are included.
- External references must use general-mode addressing so any of the referenced procedures can be put in a shareable image without requiring changes to the calling program. (See Section 5.2.5.)
- Procedures cannot require their callers to pass dynamic string descriptors. (See Section 4.2.)
- Some procedure interface specifications retain state information from one call to the next, even though the procedures are not resource allocating. The interface specification uses one of the following techniques to permit sequences of calls from independent parts of a program.

These techniques either eliminate the use of static storage or overcome its limitations (in order of decreasing preference):

1 The interface specification consists of a sequence of calls to a set of one or more procedures—the first procedure allocates and returns (as an output argument to the calling program) one of the following:

   - The address of heap storage
   - Some other processwide identifying value

   This argument is passed to the other procedures explicitly by the calling program, and the last procedure deallocates any heap storage or processwide identifying value.

2 The procedure's caller allocates all storage and passes the address on each call.

3 The interface specification consists of a single call, where the calling program passes the address of one or more action routines and arguments to be passed to them. The procedure calls the action routine(s) during its execution. Results are retained by the procedure across calls to the action routine(s). (No static storage used.)

4 The interface specification consists of a sequence of calls to a set of one or more procedures. The first procedure, among other things, saves the contents of any still active static storage on a push down stack in heap storage, and the last procedure, among other things, restores the old contents of static storage. Thus, static storage is made available for implicit arguments to be passed from one procedure to the next in the sequence of calls (unknown to the calling program). However, if an exception can occur anywhere in the sequence, the calling program must establish a condition handler that calls the last procedure in the event of a stack unwind (to restore the old contents of static storage).

A.3.2 Initialization

- If a procedure requires initialization once for each image activation, it is done without the caller's knowledge by one of the following:

  1 Initializing at compile time.

  2 Initializing at link time.

  3 Adding a dispatch address to PSECT LIB$INITIALIZE.

  4 Testing and setting a statically allocated first-time flag on each call.

  (See Section 3.2.)

- A procedure must not use LIB$INITIALIZE to establish a condition handler before the main program is called if its action might conflict with that of other condition handlers established before the main program. (See Section 3.2.3.)

A.3.3 Reporting Exception Conditions

- A procedure must not print error or informational messages either directly or by calling the $PUTMSG system service. It must either return a condition value in R0 as a function value, or call LIB$SIGNAL or LIB$STOP to output all messages. (LIB$SIGNAL and LIB$STOP may be called either directly or indirectly.) (See Section 2.4.)

A.3.4 AST Reentrancy

- To be AST-reentrant, a procedure must execute correctly while allowing any procedure (including itself) to be called between any two instructions. The other procedure can be an AST-level procedure, a condition handler, or another AST-reentrant procedure. (See Section 3.3.)
- A procedure that uses no static storage and calls only AST-reentrant procedures is automatically AST-reentrant. (See Section 3.3.3.)

- If a procedure uses static storage, it must use one of the following methods to be called from AST and non-AST levels :

  1 Perform access and modification of the data base in a single uninterruptable instruction. This can be done only from VAX MACRO, and emulated instructions are not allowed. (See Section 3.3.4.1.)

  2 Detect concurrency of data base access with "test and set" instructions at each access of the data base. (See Section 3.3.4.2.)

  3 Keep a call-in-progress count incremented upon entry to the procedure and decremented upon return. (See Section 3.3.4.3.)

  4 Disable AST interrupts on entry to the procedure and restore the state of the AST enables on return. (See Section 3.3.4.4.)

- If a procedure performs I/O from the AST level by calling VAX RMS $GET and $PUT system services, it must check for the record stream active error status (RMS$_RSA). If this error is encountered, the procedure issues the $WAIT system service and then retries the $GET or $PUT system service. (See Section 3.3.5.)

- A procedure should not depend on AST interrupts being disabled to execute correctly if there are other coding methods available. For example, RMS completion routines are implemented via ASTs and will not work if ASTs are disabled. (See Section 3.3.)

A.3.5 Resource Allocation

- A procedure should not allocate static storage unless it:

  1 Is a processwide, resource-allocating procedure, or

  2 Must retain results for implicit inputs on subsequent invocations.

- Timing procedures and resource allocation procedures should make statistics available for performance evaluation and debugging by providing the entry points fac_SHOW_name and fac_STAT_name. (Optional. See Section 4.3.)
- If a procedure uses a processwide resource, it calls the appropriate resource allocating library procedure or system service to allocate the resource to avoid conflict with allocations made to other procedures. To conserve resources, a procedure that requests resource allocation:

  1 Calls the deallocation procedure before returning to the calling program, or

  2 Remembers the allocation in static storage and calls the deallocation procedure later, or

  3 Passes the responsibility for deallocation back to the calling program, or

  4 Allocates a fixed number of the resources independent of the number of times it is called

  (See Section 2.3 and Section 3.1.5.)

A.3.6 The Format and Content of Coded Modules

- Each module must be documented with a module description. (See Section 2.5.1.)
- Each procedure must be documented with a procedure description. (See Section 2.5.2.)
- When symbol definitions are to be coordinated between more than one module, (such as control blocks, procedure argument values, and completion status codes), the definitions should be centralized in a common source file. Note, however, that the modules must be written in the same language. (See Section 3.1.3.)
- Instructions and statements should be uppercase, while comments are in upper and lowercase. (Optional. See Section 3.1.4.2.)

- Optional spaces should be added to improve readability. (Optional. See Section 3.1.4.3.)
- Block comments should be used. (Optional. See Section 3.1.4.4.)
- Use symbols rather than numbers in the body of the procedure. (Optional. See Section 3.1.4.1.)
- Procedure entry point names, module names, and PSECT names must conform to the naming conventions. (See Sections 3.1.2.2, 3.1.2.4, and 3.1.2.5.)
- It is recommended that you also adhere to the naming conventions in chosing names for facilities and files. (Optional. See Sections 3.1.2.1 and 3.1.2.3.)

A.3.7 Shareable Images

- A procedure's code is position independent. All references to external locations such as VAX/VMS System Service entry points, use general addressing mode. (Optional. See Section 5.2.5.)

A.3.8 Upward Compatibility

- When a new version of a procedure replaces an existing library procedure, all new arguments must be added at the end of the call sequence and made optional to maintain upward compatibility. (Optional. See Sections 2.2.5 and 6.1.)
- A procedure's entry points are vectored using a separate MACRO module containing TRANSFER declarations. (Optional. See Section 5.2.2.)
- A procedure's code and data is position independent. (See Section 3.1.1.)

B Argument Characteristics

Each explicit argument is defined in terms of VMS usage, data type, access mechanism and passing mechanism. These four argument attributes are described in the sections below.

B.1 VMS Usage

The VMS usage entry indicates the abstract data structure of the argument. Table B–1 contains a list of the VMS data structures.

Table B–1 VMS Data Structures

Data Structure	Definition
access_bit_names	Homogeneous array of 32 quadword descriptors; each descriptor points to the name of one of the 32 bits in an access mask.
access_mode	Unsigned byte denoting a hardware access mode. This unsigned byte can take four values: 0 specifies kernel mode; 1, executive mode; 2, supervisor mode; and 3, user mode.
address	Unsigned longword denoting the virtual memory address of either data or code, but not of a procedure entry mask (which is of type "procedure").

Table B–1 (Cont.) VMS Data Structures

Data Structure	Definition
address_range	Unsigned quadword denoting a range of virtual addresses, which identify an area of memory. The first longword specifies the beginning address in the range; the second longword specifies the ending address in the range.
arg_list	Procedure argument list consisting of one or more longwords. The first longword contains an unsigned integer count of the number of successive, contiguous longwords, each of which is an argument to be passed to a procedure by means of a VAX CALL instruction.
ast_procedure	Unsigned longword integer denoting the entry mask to a procedure to be called at AST level. (Procedures that are not to be called at AST level are of type "procedure".)
boolean	Unsigned longword denoting a boolean truth value flag. This longword may have only two values: 1 (true) and 0 (false).
byte_signed	This VMS data type is the same as the data type "byte (signed)"in Table B–2.
byte_unsigned	This VMS data type is the same as the data type "byte integer (unsigned)" in Table B–2.
channel	Unsigned word integer that is an index to an I/O channel.

Table B–1 (Cont.) VMS Data Structures

Data Structure	Definition
char_string	String of from 0 to 65,535 8-bit characters. This VMS data type is the same as the data type "character string" in Table B–2.
complex_number	One of the VAX standard complex floating-point data types.
cond_value	Unsigned longword integer denoting a condition value (that is, a return status or system condition code), which is typically returned by a procedure in R0.
context	Unsigned longword that is used by a called procedure to maintain position over an iterative sequence of calls. It is usually initialized by the caller, but thereafter manipulated by the called procedure.
date_time	64-bit unsigned, binary integer denoting a date and time as the number of elapsed 100-nanosecond units since 00:00 o'clock, November 17, 1858. This VMS data type is the same as the data type "absolute date and time" in Table B–2.
device_name	Character string denoting the name of a device. It can be a logical name, but if it is, it must translate to a valid device name.
ef_cluster_name	Character string denoting the name of an event flag cluster. It can be a logical name, but if it is, it must translate to a valid event flag cluster name.

Table B–1 (Cont.) VMS Data Structures

Data Structure	Definition
ef_number	Unsigned longword integer denoting the number of an event flag.
exit_handler_block	Variable-length structure denoting an exit handler control block.
fab	Structure denoting an RMS file access block.
file_protection	Unsigned word that is a 16-bit mask that specifies file protection. The mask contains four 4-bit fields, each of which specifies the protection to be applied to file access attempts by one of the four categories of user: from the rightmost field to the leftmost field, (1) system users, (2) the file owner, (3) users in the same UIC group as the owner, and (4) all other users (the world). Each field specifies, from the rightmost bit to the leftmost bit: (1) delete access, (2) execute access, (3) write access, (4) read access. Set bits indicate that access is denied.
floating_point	One of the VAX standard floating-point data types.
function_code	Unsigned longword specifying the exact operations a procedure is to perform. This longword has two word-length fields: the first field is a number specifying the major operation; the second field is a mask or bitvector specifying various suboperations within the major operation.

Table B–1 (Cont.) VMS Data Structures

Data Structure	Definition
io_status_block	Quadword structure containing information returned by a procedure that completes asychronously. The information returned varies depending on the procedure.
item_list_2	Structure that consists of one or more item descriptors and that is terminated by a longword containing 0. Each item descriptor is a 2-longword structure that contains three fields.
item_list_3	Structure that consists of one or more item descriptors and that is terminated by a longword containing 0. Each item descriptor is a 3-longword structure that contains four fields.
item_quota_list	Structure that consists of one or more quota descriptors and that is terminated by a byte containing a value defined by the symbolic name PQL$_LISTEND. Each quota descriptor consists of a 1-byte quota name followed by an unsigned longword containing the value for that quota.
lock_id	Unsigned longword integer denoting a lock identifier. This lock identifier is assigned by the lock manager facility to a lock when the lock is granted.

Table B–1 (Cont.) VMS Data Structures

Data Structure	Definition
lock_status_block	Structure into which the lock manager facility writes status information about a lock. A lock status block always contains at least two longwords: the first word of the first longword contains a status code; the second word of the first longword is reserved to DIGITAL; and the second longword contains the lock identifier (VMS type "lock_id".)
lock_value_block	16-byte block that the lock manager facility includes in a lock status block if the user requests it. The contents of the lock value block are user-defined and are not interpreted by the lock manager facility.
logical_name	Character string of from 1 to 255 characters that identifies a logical name or equivalence name to be manipulated by VMS logical name system services. Logical names that denote specific VMS objects have their own VMS types: for example, a logical name identifying a device has the VMS type "device_name".
longword_signed	This VMS data type is the same as the data type "longword integer (signed)" in Table B–2.
longword_unsigned	This VMS data type is the same as the data type "longword (unsigned)" in Table B–2.

Table B–1 (Cont.) VMS Data Structures

Data Structure	Definition
mask_byte	Unsigned byte wherein each bit is interpreted by the called procedure. A mask is also referred to as a set of "flags" or as a "bitmask".
mask_longword	Unsigned longword wherein each bit is interpreted by the called procedure. A mask is also referred to as a set of "flags" or as a "bitmask".
mask_quadword	Unsigned quadword wherein each bit is interpreted by the called procedure. A mask is also referred to as a set of "flags" or as a "bitmask".
mask_word	Unsigned word wherein each bit is interpreted by the called procedure. A mask is also referred to as a set of "flags" or as a "bitmask".
null_arg	Unsigned longword denoting a "null argument." A "null argument" is an argument whose only purpose is to "hold a place" in the argument list.
octaword_signed	This VMS data type is the same as the data type "octaword integer (signed)" in Table B–2.
octaword_unsigned	This VMS data type is the same as the data type "octaword (unsigned)" in Table B–2.
page_protection	Unsigned longword specifying page protection to be applied by the VAX hardware. Protection values are specified using bits 0 to 3; bits 4 to 31 are ignored.

Table B–1 (Cont.) VMS Data Structures

Data Structure	Definition
procedure	Unsigned longword denoting the entry mask to a procedure that is not to be called at AST level. (Arguments specifying procedures to be called at AST level have the VMS type "ast_procedure".)
process_id	Unsigned longword integer denoting a process identifier (PID). This process identifier is assigned by VMS to a process when the process is created.
process_name	Character string that specifies the name of a process.
quadword_signed	This VMS data type is the same as the data type "quadword integer (signed)" Table B–2.
quadword_unsigned	This VMS data type is the same as the data type "quadword (unsigned)" in Table B–2.
rights_holder	Unsigned quadword specifying a user's access rights to a system object. This quadword consists of two fields: the first is an unsigned longword identifier (VMS type "rights_id") and the second is a longword bitmask wherein each bit specifies an access right.
rights_id	Unsigned longword denoting a rights identifier, which identifies an interest group in the context of the VMS security environment. This rights environment may consist of all or part of a user's User Identification Code (UIC).

Table B–1 (Cont.) VMS Data Structures

Data Structure	Definition
rab	Structure denoting an RMS record access block.
section_id	Unsigned quadword denoting a global section identifier. This identifier specifies the version of a global section and the criteria to be used in matching that global section.
section_name	Character string denoting a global section name. This character string can be a logical name, but it must translate to a valid global section name.
system_access_id	Unsigned quadword that denotes a system identification value that is to be associated with a rights database.
time_name	Character string specifying a time value in VMS format.
uic	Unsigned longword denoting a User Identification Code (UIC).
user_arg	Unsigned longword denoting a user-defined argument. This longword is passed to a procedure as an argument, but the contents of the longword are defined and interpreted by the user.
varying_arg	Unsigned longword denoting a variable argument. A variable argument can have variable types, depending on specifications made for other arguments in the call.
vector_byte_signed	A homogeneous array whose elements are all signed bytes.
vector_byte_unsigned	A homogeneous array whose elements are all unsigned bytes.

Table B–1 (Cont.) VMS Data Structures

Data Structure	Definition
vector_longword_signed	A homogeneous array whose elements are all signed longwords.
vector_longword_unsigned	A homogeneous array whose elements are all unsigned longwords.
vector_quadword_signed	A homogeneous array whose elements are all signed quadwords.
vector_quadword_unsigned	A homogeneous array whose elements are all unsigned quadwords.
vector_word_signed	A homogeneous array whose elements are all signed words.
vector_word_unsigned	A homogeneous array whose elements are all unsigned words.
word_signed	This VMS data type is the same as the data type "word integer (signed)" in Table B–2.
word_unsigned	This VMS data type is the same as the data type "word (unsigned)" in Table B–2.

B.2 Data Type

An argument's data type indicates the VAX data type that must be used for the argument.

Table B–2 contains the data types allowed by the VAX Procedure Calling and Condition Handling Standard.

Table B–2 VAX Standard Data Types

Data Type	Symbolic Code
Absolute date and time	DSC$K_DTYPE_ADT
Byte integer (signed)	DSC$K_DTYPE_B
Bound label value	DSC$K_DTYPE_BLV
Bound procedure value	DSC$K_DTYPE_BPV
Byte (unsigned)	DSC$K_DTYPE_BU
COBOL intermediate temporary	DSC$K_DTYPE_CIT
D_floating	DSC$K_DTYPE_D
D_floating complex	DSC$K_DTYPE_DC
Descriptor	DSC$K_DTYPE_DSC
F_floating	DSC$K_DTYPE_F
F_floating complex	DSC$K_DTYPE_FC
G_floating	DSC$K_DTYPE_G
G_floating complex	DSC$K_DTYPE_GC
H_floating	DSC$K_DTYPE_H
H_floating complex	DSC$K_DTYPE_HC
Longword integer (signed)	DSC$K_DTYPE_L
Longword (unsigned)	DSC$K_DTYPE_LU
Numeric string, left separate sign	DSC$K_DTYPE_NL
Numeric string, left overpunched sign	DSC$K_DTYPE_NLO
Numeric string, right separate sign	DSC$K_DTYPE_NR
Numeric string, right overpunched sign	DSC$K_DTYPE_NRO
Numeric string, unsigned	DSC$K_DTYPE_NU
Numeric string, zoned sign	DSC$K_DTYPE_NZ
Octaword integer (signed)	DSC$K_DTYPE_O
Octaword (unsigned)	DSC$K_DTYPE_OU
Packed decimal string	DSC$K_DTYPE_P
Quadword integer (signed)	DSC$K_DTYPE_Q
Quadword (unsigned)	DSC$K_DTYPE_QU
Character string	DSC$K_DTYPE_T
Aligned bit string	DSC$K_DTYPE_V
Varying character string	DSC$K_DTYPE_VT

Table B–2 (Cont.) VAX Standard Data Types

Data Type	Symbolic Code
Unaligned bit string	DSC$K_DTYPE_VU
Word integer (signed)	DSC$K_DTYPE_W
Word (unsigned)	DSC$K_DTYPE_WU
Unspecified	DSC$K_DTYPE_Z
Procedure entry mask	DSC$K_DTYPE_ZEM
Sequence of instruction	DSC$K_DTYPE_ZI

B.3 Access Mechanism

The argument access entry describes the way in which the called routine accesses the data specified by the argument. The following three methods of access are the most common. (In the explanation below, the formal argument is the procedure argument from the point of view of the called procedure, and the actual argument is the procedure argument from the point of view of the procedure's caller.)

1 Read only. The formal argument is a constant. The associated actual argument may only be read; it may not be written to or modified.

2 Write only. The formal argument is a variable. The value of the associated actual argument is written into the variable by the called procedure.

3 Modify. The formal argument is a variable. The value of this variable is written by the procedure's caller and may be read and/or modified by the called procedure.

The following is a complete list of the access types allowed by the VAX Procedure Calling and Condition Handling Standard.

- Read only
- Write only
- Modify
- Function call (before return)
- JMP after unwind

- Call after stack unwind
- Call without stack unwind

B.4 Passing Mechanisms

The argument passing mechanism is the way in which an argument specifies the actual data to be used by the called routine. There are three types of passing mechanism.

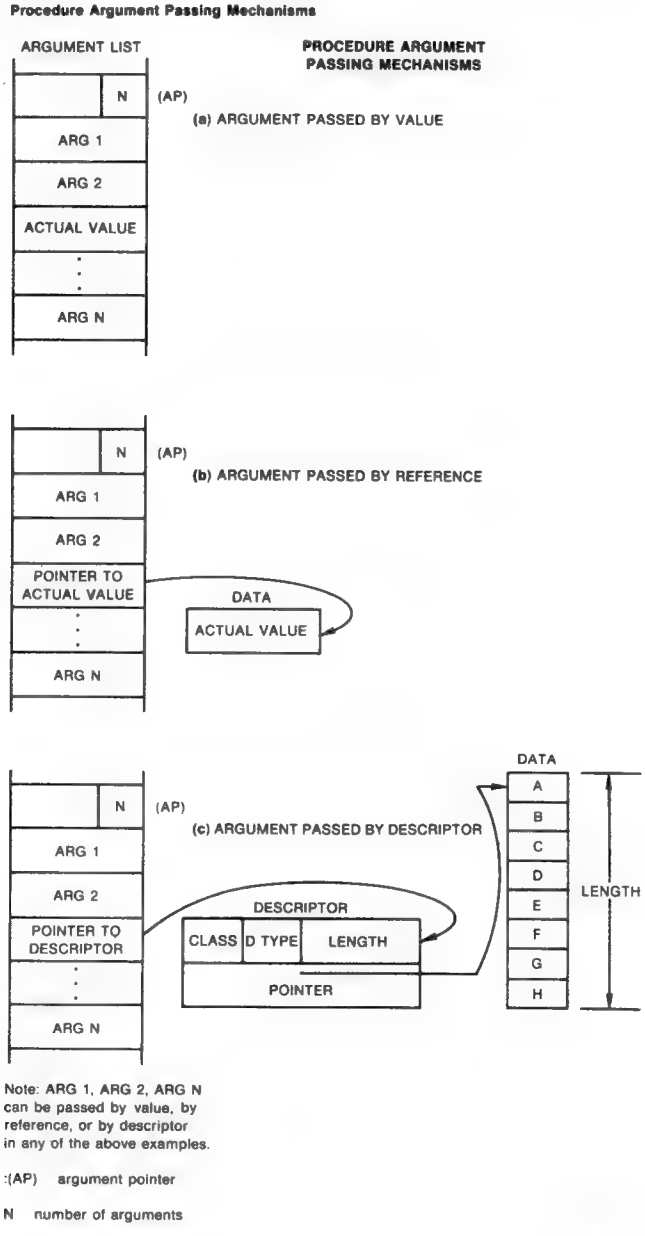
1 By value. When the longword argument in the argument list contains the actual data to be used by the routine, the actual data is said to be passed to the routine "by value". Note that since an argument is only one longword in length, only data that can be represented in one longword can be passed by value.

2 By reference. When the longword argument in the argument list contains the address of the data to be used by the routine, the data is said to be passed "by reference".

3 By descriptor. When the longword argument in the argument list contains the address of a descriptor, the data is said to be passed "by descriptor". A descriptor consists of two or more longwords (depending on the class of descriptor used), which describe the location, length, and data type of the data to be used by the called routine.

Figure B–1 illustrates the three passing mechanisms.

Figure B–1 Procedure Argument Passing Mechanisms



Note: ARG 1, ARG 2, ARG N can be passed by value, by reference, or by descriptor in any of the above examples.

:(AP) argument pointer

N number of arguments

ZK-1962-81

Table B–3 contains the passing mechanisms allowed by the VAX Procedure Calling and Condition Handling Standard:

Table B–3 VAX Standard Passing Mechanisms

Passing Mechanism	Descriptor Code
By value	
By reference	
By reference, array reference	
By descriptor	
By descriptor, fixed-length	DSC$K_CLASS_S
By descriptor, dynamic string	DSC$K_CLASS_D
By descriptor, array	DSC$K_CLASS_A
By descriptor, procedure	DSC$K_CLASS_P
By descriptor, decimal string	DSC$K_CLASS_SD
By descriptor, noncontiguous array	DSC$K_CLASS_NCA
By descriptor, varying string	DSC$K_CLASS_VS
By descriptor, varying string array	DSC$K_CLASS_VSA
By descriptor, unaligned bit string	DSC$K_CLASS_UBS
By descriptor, unaligned bit array	DSC$K_CLASS_UBA
By descriptor, string with bounds	DSC$K_CLASS_SB
By descriptor, unaligned bit string with bounds	DSC$K_CLASS_UBSB

Index

F

I

J

L

M

N

O

P

R

S

T

U

V

W

READER'S COMMENTS

Note: This form is for document comments only. DIGITAL will use comments submitted on this form at the company's discretion. If you require a written reply and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Did you find this manual understandable, usable, and well organized? Please make suggestions for improvement.

_____

_____

_____

_____

_____

_____

Did you find errors in this manual? If so, specify the error and the page number.

_____

_____

_____

_____

_____

_____

Please indicate the type of user/reader that you most nearly represent:

- ☐ Assembly language programmer
- ☐ Higher-level language programmer
- ☐ Occasional programmer (experienced)
- ☐ User with little programming experience
- ☐ Student programmer
- ☐ Other (please specify) _____

Name _____ Date _____

Organization _____

Street _____

City _____ State _____ Zip Code _____
or Country

Do Not Tear - Fold Here and Tape

digital

No Postage Necessary if Mailed in the United States

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO.33 MAYNARD MASS.

POSTAGE WILL BE PAID BY ADDRESSEE

SSG PUBLICATIONS ZK1-3/J35
DIGITAL EQUIPMENT CORPORATION
110 SPIT BROOK ROAD
NASHUA, NEW HAMPSHIRE 03062-2698

Do Not Tear - Fold Here

Cut Along Dotted Line